Dynamically Distributed 2D Game Physics

Michael Söllinger

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im September 2015

© Copyright 2015 Michael Söllinger

This work is published under the conditions of the *Creative Commons License Attribution–NonCommercial–NoDerivatives* (CC BY-NC-ND)—see http://creativecommons.org/licenses/by-nc-nd/3.0/.

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, September 28, 2015

Michael Söllinger

Contents

Abstract

The present thesis engages in the creation of a dynamically distributed 2D game physics simulation targeting low-performance microcomputers. To overcome said performance restrictions the computationally demanding task of physics simulation is distributed among multiple processing nodes and dynamically load balanced during runtime. The nodes of this distributed system connect to an enclosed *LAN* via *Ethernet*.

Current game programming techniques and patterns are taken into account during development. Special focus lies on the creation of an efficient network protocol with minimized overhead. A preexisting third-party 2D physics engine handles the actual simulation of the objects in a game's virtual environment. Clustering algorithms are compared, and a suitable one is chosen, to allow distribution of these dynamic objects. The resulting clusters are load balanced among the processing nodes to evenly utilize processor capacity and optimize overall performance.

The implemented system is evaluated by comparing the results of various test scenarios, which are executed multiple times on different hardware configurations.

Kurzfassung

Die vorliegende Arbeit beschäftigt sich mit der Erstellung einer dynamisch verteilten Physik Simulation für 2D-Spiele auf leistungsschwachen Mikrocomputern. Um die Einschränkungen durch die geringe Leistung zu umgehen, wird die anspruchsvolle Aufgabe der Physik Simulation auf mehrere dieser Computer verteilt. Das geschieht mittels dynamischer Lastverteilung über die gesamte Laufzeit der Anwendung. Die einzelnen Knoten dieses verteilten Systems sind über *Ethernet* zu einem *LAN* verbunden.

Während der Entwicklung werden aktuelle Techniken und Muster aus dem Bereich der Spieleentwicklung berücksichtigt. Ein besonderer Fokus liegt hierbei auf der Erstellung eines effizienten Netzwerk-Protokolls. Für die eigentliche physikalische Simulation der Objekte, in der virtuellen Umgebung des Spiels, wird eine existierende 2D-Physik-Engine verwendet. Um eine Verteilung der Simulation zu ermöglichen werden die dynamischen Objekte mit einem passenden Algorithmus in Cluster zusammengefasst. Die resultierenden Cluster werden dann so auf die Rechen-Knoten verteilt, dass eine gleich mäßige Auslastung und optimale Leistungsfähigkeit gewährleistet ist.

Die Evaluierung des implementierten Systems erfolgt durch den Vergleich der Ergebnisse verschiedener Test-Szenarios, welche auf mehreren unterschiedlichen Hardware Konfigurationen ausgeführt werden.

Chapter 1

Introduction

1.1 Motivation

Since its introduction, the *Raspberry Pi* and similar inexpensive low-performance microcomputers have proven highly successful and are used widely in different research fields. An area where they may not seem ideal at first is game development because of the computationally highly demanding real-time requirements of certain game types. However, low power consumption and low heat emission make them a logic alternative to more powerful fan cooled computers when used in closed arcade cases. To overcome the performance bottleneck, a distributed game engine running on multiple microcomputers is an easily extensible solution.

Game physics simulation became the second most demanding area after visual representation regarding required processing power. To achieve high scalability within the described system, physics simulation could be distributed among several processors in addition to the more obvious ideas like separating rendering and logic or artificial intelligence.

Further, a distributed physics simulation targeting microcomputers also provides an inexpensive testing ground before taking the idea to more expensive high-performance hardware.

1.2 Research Question

Based on the premise described in the previous section following research question was formulated: Are dynamic clustering algorithms applicable to distribute game objects to allow distributed simulation of game physics? Which algorithms are suited for initial distribution and dynamic update during runtime? What load balancing strategies are feasible to dynamically allocate processing resources to simulate the clustered objects?

1.3 Objectives

This thesis engages in the development of distributed 2D game physics simulation built on top a distributed game engine. The intended goal is to answer the research questions based on the evaluation of example use cases that facilitate this distributed engine.

1.4 Thesis Structure

The thesis starts in Chapter 2 with an introduction to the existing literature that forms the foundation for a distributed 2D game physics system. This basic knowledge includes general game engine design, game physics, clustering algorithms and load balancing. After that, the actual implementation of the system is presented in Chapter 3, including a distributed game engine, distributed physics simulation and a test application for actual use cases. Results obtained by running the test application in various usage scenarios are then displayed, compared and discussed in Chapter 4. A final verdict if the set objective was achieved and an outlook for possible improvements is shown in the concluding Chapter 5.

Chapter 2

Foundation

This chapter details the foundation on which the thesis project builds. A general introduction to game engine development is followed by a detailed look at network programming in games. These two topics form the core on which the other parts build. The next section looks at the simulation of game physics and the concept of a physics engine. After that, clustering algorithms are discussed. The last section of the chapter details load balancing techniques.

2.1 Game Engine

In today's professional game development, the actual game is created by facilitating a pre-existing or custom-built game engine. Those engines can be general purpose targeting multiple game genres or offer features tailored to a single game genre. They include modules for input handling, output (3D/2D graphics, sound, force feedback) and physics/dynamics simulation. In other words all the game code that does not directly specify the games behaviour or environment data [13].

As shown in Figure 2.1 the engine is the layer that sits directly above the operating system and graphics API[1]. It provides a higher level abstraction of these resources to the game code. In the later described engine, the network code is more integrated into the engine due to its distributed nature (see Chapter 3 for details).

This section gives a short introduction to game engine techniques and game programming patterns used as a foundation for the distributed physics simulation.

[1]Application Programming Interface.

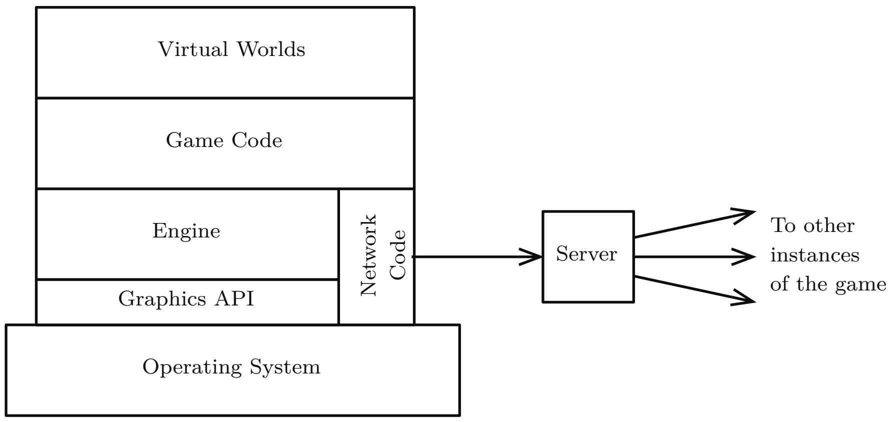


Figure 2.1: Modular game engine structure [13].

Algorithm 2.1: Basic game loop.

```
1: RunGame
2:     while true do
3:         ProcessUserInput()
4:         UpdateGameState()
5:         Render()
6:     end while
7: end
```

2.1.1 Game Loop

At the core of almost any game engine lies the game loop [15, Chapter 9]. A basic implementation of a game loop is an infinite loop. Within the loop's body user input is processed, the game state is updated, and the current state's representation is rendered to the screen (see Alg. 2.1).

An actual implementation of a game loop is more complex than this simple concept. For Example different update rates for subsystems like rendering and physics simulation should be supported.

2.1.2 Event Queue

An event queue (or event loop, message dispatcher, message pump, ...) is a programming construct that decouples the sending of a message or event from when it is processed [33]. It is also widely used in GUI systems for user input handling. In a game engine, an event queue can be used to decouple different sub-systems [15, Chapter 15]. Figure 2.2 shows a *UML*[2] diagram of an example event queue. In contrast to the observer pattern [9, Chapter 17]

[2]Unified Modeling Language: http://www.uml.org/.

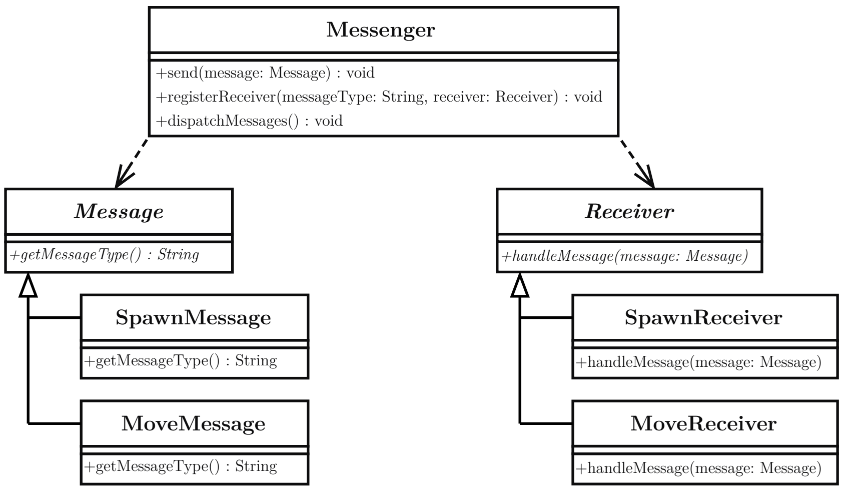

Messenger
+send(message: Message) : void
+registerReceiver(messageType: String, receiver: Receiver) : void
+dispatchMessages() : void
Message
+getMessageType() : String
Receiver
+handleMessage(message: Message)
SpawnMessage
+getMessageType() : String
MoveMessage
+getMessageType() : String
SpawnReceiver
+handleMessage(message: Message)
MoveReceiver
+handleMessage(message: Message)

Figure 2.2: UML diagram of an example event queue implementation. Events are called messages in this case.

which also allows decoupling of message sender and receiver, an event queue also allows to decouple message creation and handling in time.

2.1.3 Service Locator

The service locator pattern allows further decoupling by providing a central access point to game services (e.g.: render system, log system, ...) [15, Chapter 16].

Clients can access these services without knowing about their actual implementation and services can be designed without considering global access to them.

2.2 Network Programming in Games

Networking in games is needed when the gameplay or parts of it are distributed among several computers. In many cases, these games are multiplayer where each player runs the game on a separate machine. Of course not all multiplayer games are networked [3, Section 2.1]. An efficient implementation of a game's network layer is necessary due to the fact that network resources are limited (see Section 2.2.1). This is especially true for real time games. Since the thesis focuses on a distributed simulation in an local area network this section limits most details to this type of network.

2.2.1 Resource Limitations

In the design of distributed games, three resource limitations of computer networks have to be considered. They are network bandwidth, network latency and the required computational power for handling the network traffic [24, 25].

Bandwidth

Available Bandwidth of a network describes the maximum data throughput of said network. From the applications perspective the amount of sent packages and their size define the needed bandwidth. Bandwidth requirement also increases with the number of recipients. *Broadcast* and *Multicast* are techniques that may be used to prevent sending of redundant data and reduce bandwidth in cases with multiple recipients.

Due to the real-time requirement in game development bandwidth concerns are secondary to latency.

Latency

Network latency specifies the delay between sending and receiving of a message. It can be measured either in one way or in both ways as *round-trip time*. In real-time games, latency is the most significant restriction to take into account.

Sources of Latency are [3, Section 5.2]:

- *Propagation delay*: Is the time a packet needs to travel from sender and receiver. It is lower than or in the best case equal to the speed of light. The greater the distance between sender and receiver the larger the delay.
- *Serialisation delay*: This delay occurs at most link layers when the frames are broken up into sequences of bytes which are then sent one bit at a time. It depends on the speed of the link and the length of the packet being sent.
- *Queuing delay*: Occurs at routers if packets arrive faster than the router can process them. For example, this is possible due to a capacity overload during a burst transmission. It can lead to delays up to several seconds.

The actually acceptable latency differs depending on the type of games [7]. Studies showed that in first person shooters for example latency from 75ms to 100ms was already noticeable for players [5]. Latency above 100ms leads to distinctly lower accuracy. The authors of [20] state that even latency of only 60ms was distracting for some players. In real-time strategy games, latency of several seconds is acceptable without noticing and influencing the performance of a player [21]. Racing games again tolerate only very small latency values. First influences on player's performance show already at a latency of only 50ms. Values above 100ms should be omitted because they lead to unrealistic driving behaviour [16].

Computational Power

Besides the two restrictions of the actual network, also the processing power of the computers running the distributed system has to be considered. It is often overlooked, but can require a noticeable amount of processing power, especially on less powerful systems when dealing with a large number of connections [22].

2.2.2 Network Protocols

The computers running the distributed engine are connected via Ethernet. The most common and widely supported protocols are TCP and UDP, which both build on top of the Internet Protocol (IP) [17]. Each has its benefits and drawbacks, and the decision which one to use depends mainly on the requirements of the particular application [35].

Transmission Control Protocol (TCP)

TCP is a connection-oriented transport protocol [18]. Prior to the actual communication, a connection has to be established. During data transmission, the protocol guarantees ordered and reliable delivery of data. To achieve this the receiver queues all packets and if one got lost asks the sender to retransmit it. This mechanism has the downside that it delays the transmission, especially in the case of packet loss. The header on each sent data stream contains metadata needed to provide the described reliability. This leads to a typical header size of 20 Bytes. These facts make the protocol not that well suited for the real-time requirements of video games.

User Datagram Protocol (UDP)

In contrast to TCP, UDP is a connection-less protocol [19]. It does not provide any functionality for a reliable transmission which means lost packages are not resent automatically. However, this has the benefit that a header of 8 Bytes size is needed and no waiting time for lost packages occurs. If needed, features of TCP can be implemented on top of UDP [34, 36].

2.2.3 Message Transmission

Messages in a local area network can either be transmitted as unicast, broadcast or multicast. These transmission methods are implemented in the IP protocol. Unicast transmission sends a message only to a single receiver.

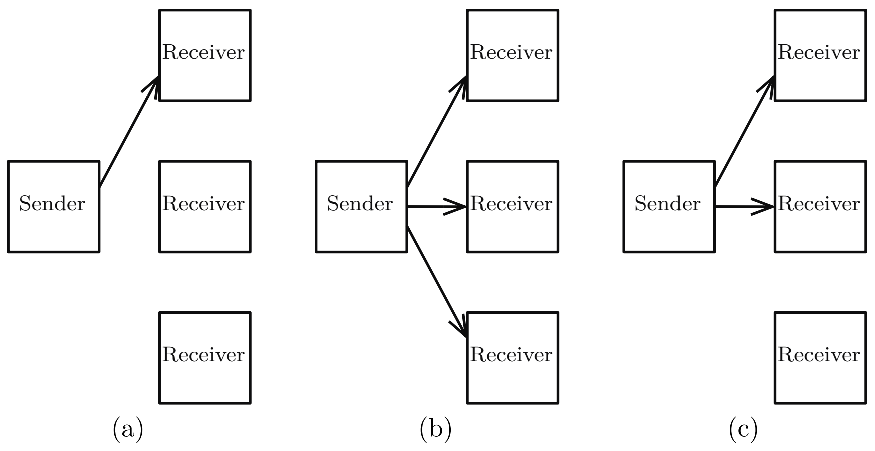


Figure 2.3: Message sending in unicast transmission (a), broadcast transmission (b) and multicast transmission (c) in comparison.

Broadcast, in contrast, transmission sends a message to all members of the network. A multicast transmission sends messages only to specific recipients. A schematic comparison in shown in Figure 2.3.

2.3 Game Physics

Physics simulation in video games is needed when games try to simulate virtual worlds based on the physics of the real world. For example, this may be wanted to create a believable game world in an action or strategy game, simulate vehicles a racing or flight simulation or allow physics based puzzles in certain puzzle games.

When speaking of game physics this thesis, like most literature on physics in games, means classical mechanics. Other parts of academic physics are only simulated if they are an important element of the gameplay. Or optics, for example, which plays into the real-time graphics part of game development.

Since the first games have been created physical simulation has been a part of them. With the increase of processing power, the physics simulation became more sophisticated. Beginning with simple bullet ballistics in early games and leading to destructible objects and ragdoll bodies[3], which simulate the human skeleton to allow more realistic trips, falls and death scenes in today's blockbuster games [14, Chapter 1].

In most games, simulation speed is more important than accuracy. Due to this requirement game physics are only an approximation and simplification of real world physics. Depending on the type of game such a simplification

[3]https://en.wikipedia.org/wiki/Ragdoll_physics.

could be to simulate physics only in two dimensions instead of three. This, for example, is the case in the distributed physics discussed in this thesis.

2.3.1 Physics Engine

Initially, each physical effect was directly programmed into a game's code as needed. The more complex physics in games became, the more effects had to be added separately. This lead to the idea of creating more general physics simulations in games [14, Chapter 1].

Like a graphics engine for the rendering part, a physics engine handles the physical simulation of the game world. From the developer's view, this engine is a black box, which does all the calculations needed to simulate the physics. In comparison to directly implemented physics, the developer has to define the objects and the forces that are applied to them.

Advantages

There are two key benefits when implementing a physics engine in games. First it is reusable in more than one game. This way a general purpose physics engine leads to time saving.

Second there is the quality of the physics simulation. When using multiple effects, all created on their own, the combination of them can lead to visible errors that destroy the immersion in a game. In a general purpose engine, combined effects on objects happen automatically. Improved quality, of course, can only be delivered if the engine was developed carefully [14, Chapter 1].

Disadvantages

The main problem of a general purpose physics engine is the processing power needed for simulation. In games that only require specific physical effects performance may be better if some restricting assumptions can be made [14, Chapter 1].

2.3.2 Physics Islands

The concept of an island was created to parallelize simulation of physics. An *island* is a group of bodies that cannot be pulled apart of each other and because of that have to be simulated together. On the other hand, this means that each *island* can be solved independently on a separate processing unit [28]. In physics engines *island* creation, distribution among threads and solving happens encapsulated within the simulation step and usually cannot be accessed from the outside. The creation of *islands* takes place after pairs of the objects in contact have been calculated [29].

2.4 Clustering Algorithms

Clustering Algorithms are used to organise spatial data into relatively homogenous groups. In the distributed game physics simulation these clusters are used to form separated physical islands, which then can be processed on their own.

2.4.1 Approaches

Clustering algorithms may be divided into two basic types: Partitioning and hierarchical algorithms [12, Section 1.3].

- *Partitioning algorithms*, as the name suggest, construct a partition of a dataset of n objects into a set of k clusters. Clusters are represented by the gravitational center of the cluster (*k-means*) or by an object of the cluster located near its center (*k-medoid*). The shape of all found clusters is convex, which may be a restriction that prevents optimal cluster discovery.

  The number of clusters k is a required input parameter, which means that it has to be known in advance. For the arbitrary distribution of simulated physical bodies in a game, this is not true. Therefore, partitioning algorithms are not suited for application in distributed game physics.
- *Hierarchical algorithms* create hierarchical decomposition of a given dataset. This decomposition is represented by a tree that iteratively splits D into smaller subsets until there is only one object left in each subset. Each node of the tree represents a cluster of the dataset.

  The tree can either be created bottom-up from the leaves to the root (*agglomerative approach*) or top-down from the root to the leaves (*divisive approach*).

  An advantage over partitioning algorithms is that the number of clusters (k) needs not to be known in advance. However, a termination condition has to be defined to indicate when the merge or division process should stop, and the clustering is finished. One example of a termination condition in the agglomerative approach is the critical distance D_{min} between the clusters of the dataset. Deriving the termination condition is a main problem with hierarchical algorithms. *Ejcluster* is an algorithm that automatically calculates the termination condition, this, however, comes with the computational cost of $O(n^2)$ [8].

Additional to these older categorizations there are also density-based clustering algorithms. In density-based clustering, clusters are defined as regions in the data space in which objects are dense. These clusters are separated by regions of low density (noise). The clusters may have arbitrary shape and

Algorithm 2.2: DBSCAN algorithm [8]. *NOISE* and *UNCLASSIFIED* are markers to distinguish data points which are not part of a cluster or have not yet been classified. When the algorithm is finished all points have either a cluster id or are marked as *NOISE*.

```
1:  DBSCAN($\mathcal{P}, \epsilon, p_{min}$)           ▷ The set of points $\mathcal{P}$ is UNCLASSIFIED.
2:      $c \leftarrow 0$                                       ▷ $c$ is the next cluster id.
3:      for all $p \in \mathcal{P}$ do
4:          if $p_c$ = UNCLASSIFIED then        ▷ cluster id of $p$ is unclassified
5:              if ExpandCluster($\mathcal{P}, p, c, \epsilon, p_{min}$) then        ▷ see Alg. 2.3.
6:                  $c \leftarrow c + 1$                           ▷ Increment cluster id $c$.
7:              end if
8:          end if
9:      end for
10: end
```

the points inside each cluster may be arbitrarily distributed [26, Chapter 8].

2.4.2 DBSCAN

DBSCAN is a *density-based* algorithm for discovering clusters in spatial databases with noise. To count as a cluster for each point, the neighbourhood of a given radius has to contain at least a minimum number of points. In other words, the density of a neighbourhood has to exceed some threshold. The algorithm supports any distance function appropriate to the application. So in the case of clustering 2D physics bodies the Euclidian distance in 2D can be used. A basic version of *DBSCAN* can be seen in Algorithm 2.2. It is described in full detail in [8].

In comparison to the partitioning approach, the density based *DBSCAN* clustering offers improved handling of clusters of different size, convex clusters and clusters of different density (see Fig. 2.4).

2.4.3 OPTICS

OPTICS stands for *Ordering Points To Identify the Clustering Structure*. It does not produce a clustering of a data set but instead creates an ordering of the database representing its density based clustering structure [2]. It works in principle like an adapted *DBSCAN* algorithm that creates a hierarchical density based clustering. In comparison to *DBSCAN*, it allows clusters to have varying density.

2.4.4 Conclusion

As already mentioned in the previous sections clustering algorithm for use in distributed game physics should not require knowing the number of clus-

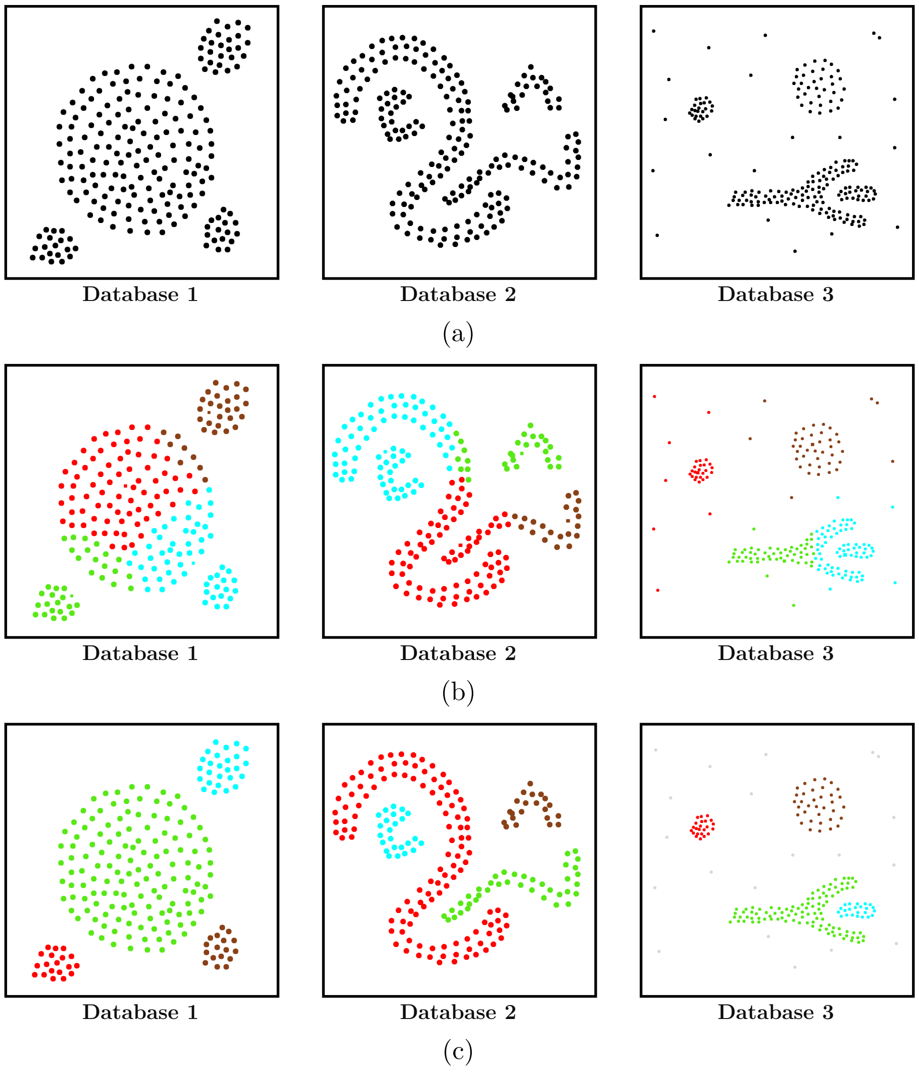


Figure 2.4: Comparison of *DBSCAN* with the *k-medoid* based *CLARANS* algorithm. The initial dataset (a), clusterings discovered by *CLARANS* (b) and the clusterings discovered by *DBSCAN* (c). In database 1 the improved handling of different sized clusters is visible, database 2 contains convex clusters which are discovered better by *DBSCAN* and database 3 shows the noise points *CLARANS* would assign to the closest medoid. Image source [8]

ters in advance. There may be some games where this number is known in advance but in general and due to player interaction with the game environment it is not known.

The noise points as yielded by the density based algorithms like *DBSCAN* can be arbitrarily distributed among the computation nodes because they are not interacting with any clusters.

Algorithm 2.3: DBSCAN expand cluster.

```
1:  ExpandCluster($\mathcal{P}, p, c, \epsilon, p_{min}$)
    Returns *true* if p is a cluster's core point or else *false*.
2:      $\mathcal{S} \leftarrow$ RegionQuery($\mathcal{P}, p, \epsilon$)    ▷ $\mathcal{S}$ contains the $\epsilon$-neighbourhood of $p$.
3:      if $|\mathcal{S}| < p_{min}$ then
4:          $p_c \leftarrow$ UNCLASSIFIED
5:          return false                              ▷ $p$ is no cluster core point.
6:      else                    ▷ All points in $\mathcal{S}$ are density reachable from $p$
7:          $p_c \leftarrow c$                                ▷ $p$'s cluster id is now $c$.
8:          Delete($\mathcal{S}, p$)                             ▷ Remove $p$ from $\mathcal{S}$.
9:          while $|\mathcal{S}| > 0$ do
10:             $q \leftarrow \mathcal{S}_0$                            ▷ Get first point $q$ of $\mathcal{S}$.
11:             $\mathcal{T} \leftarrow$ RegionQuery($P, q, \epsilon$)
12:             if $|\mathcal{S}| < p_{min}$ then
13:                 for all $r \in \mathcal{T}$ do
14:                     if $r_c =$ UNCLASSIFIED $\vee$ $r_c =$ NOISE then
15:                         if $r_c =$ UNCLASSIFIED then
16:                             Append($\mathcal{S}, r$)
17:                         end if
18:                         $r_c \leftarrow c$
19:                     end if
20:                 end for
21:             end if
22:             Delete($\mathcal{S}, q$)                         ▷ Remove $q$ from $\mathcal{S}$.
23:         end while
24:         return true
25:     end if
26: end
```

2.5 Load Balancing

The last part of the proposed dynamically distributed game physics simulation is to balance the load among the nodes which are running the engine. To allow distribution of the simulation partitions can be formed from the clusters obtained by the clustering algorithms discussed in the previous Section (2.4). Such partitioning mechanisms are required in all distributed simulations and the best of them also load-balance the partitions [4].

Work package exchange between the processing nodes can be done globally or locally when required due to load changes. In a global load balancing strategy, packages can move to any other node in the system. Local load balancing on the other hand only allows these exchanges with the defined neighbouring nodes [4].

Load balancing in distributed systems can be distinct in two classes depending on where the management is done. It can be done centralized on a single entity (master node) or decentralized on the separate entities forming the system. While the centralized approach simplifies the balancing task, it can be a performance bottleneck and increase network communication. The dezentralized method makes it hard to get a global view of the simulation and is not suited well for global load balancing [4].

2.5.1 Linear Partition Problem

When balancing load for parallel processing the *linear partition problem* arises [23]. Load has to be classified with positive integer values to fall under this domain. For balancing clusters, this can be achieved by simply taking the clusters' sizes for partitioning.

Introduction

A special case would be to balance load evenly among only two nodes, which is an example of the optimization variant of the partition problem [37]. The partition problem is one of Karp's 21 NP-complete[4] problems defined in 1972 [11].

The partition problem itself is the task to determine if a given set $\mathcal{S}$ of positive integers can be partitioned into two subsets $\mathcal{S}_1$ and $\mathcal{S}_2$ such that

$$\operatorname{sum}(\mathcal{S}_1) = \operatorname{sum}(\mathcal{S}_2), \tag{2.1}$$

where

$$\operatorname{sum}(\mathcal{S}) := \sum_{x \in \mathcal{S}} x. \tag{2.2}$$

An example set $\mathcal{S} = \{3, 1, 1, 2, 2, 1\}$ could be split into the two sets $\mathcal{S}_1 = \{1, 1, 1, 2\}$ and $\mathcal{S}_2 = \{2, 3\}$ with $\operatorname{sum}(\mathcal{S}_1) = \operatorname{sum}(\mathcal{S}_1) = 5$.

For load balancing, it is not important if there is an exact solution instead a optimal solution is needed. The optimization version of the partition problem, however, is NP-hard[5][37]. For example there exists no exact solution for $\mathcal{S} = \{2, 1, 5\}$ however an optimal solution is $\mathcal{S}_1 = \{5\}$ and $\mathcal{S}_2 = \{1, 2\}$.

Although there exists a solution to find the optimum, depending on the actual use case, an approximation may be sufficient. A naive and computationally inexpensive ($O(n \log n)$) approach for finding an approximate solution is, for example, the following greedy algorithm where the given set (e.g. $\mathcal{S} = \{2, 1, 5\}$)

1. is sorted descending (e.g. $\mathcal{S} = \{5, 2, 1\}$),
2. the first element is inserted into the first subset (e.g. $\mathcal{S}_1 = \{5\}$),

[4]https://en.wikipedia.org/wiki/NP-complete/.
[5]https://en.wikipedia.org/wiki/NP-hard/.

Algorithm 2.4: Greedy algorithm for finding an approximate optimal solution for the partition problem [37].

```
1:  PARTITION($\mathcal{S}$) $\mathcal{S}$ a set of positive integers
    Returns two sets $\mathcal{S}_1$, $\mathcal{S}_2$ containing the partitioned values.
2:      $\mathcal{S} \leftarrow \text{sort}(\mathcal{S})$                    ▷ sort $\mathcal{S}$ descending.
3:      $\mathcal{S}_1 \leftarrow \{\}$
4:      $\mathcal{S}_2 \leftarrow \{\}$
5:      for all $x \in \mathcal{S}$ do
6:          if $\text{sum}(\mathcal{S}_1) \leq \text{sum}(\mathcal{S}_2)$ then          ▷ see Eq. 2.2
7:              $\text{insert}(\mathcal{S}_1, x)$
8:          else
9:              $\text{insert}(\mathcal{S}_1, x)$
10:         end if
11:     end for
12:     return $(\mathcal{S}_1, \mathcal{S}_2)$
13: end
```

3. the next element is inserted into the second subset (e.g. $\mathcal{S}_2 = \{2\}$) and
4. all further elements are inserted into the set with the smaller sum (e.g. $\mathcal{S}_1 = \{5\}$ and $\mathcal{S}_2 = \{2, 1\}$) (see Alg. 2.4).

This algorithm is especially well suited for partitioning *DBSCAN* clusters because the noise objects, which basically are clusters containing only one single element, lead to a better approximation of the optimal result when filling the two sets. In cases where the algorithm produces a bad approximation, the other algorithms that find an optimal solution have their highest runtime complexity, which is bad for real-time applications.

Linear Partitioning Problem Algorithm

The discussed greedy algorithm can be adapted to work for the generalization of the problem, the *linear partitioning problem* (see Alg. 2.5). The modified algorithm has the same properties that make the specialized algorithm well suited for partitioning *DBSCAN* clusters in real-time application scenarios.

2.5.2 Workload Rating

A central question prior to actually balancing the workload is how it can be rated.

- For balancing clusters of objects for simulation, a simple approach could be to count the number of objects in each cluster [10]. This value is a simplified estimation of the workload.

Algorithm 2.5: Adaption of Alg. 2.4 to work for $k \geq 2$ instead of $k = 2$.

```
1: LinearPartition($\mathcal{S}, k$)    ▷ $\mathcal{S}$ a set of positive integers, $k$ number of result sets wanted
   Returns a set $\mathcal{R}$ containing $k$ sets which contain the partitioned values.
2:     $\mathcal{S} \leftarrow$ sort($\mathcal{S}$)                          ▷ sort $\mathcal{S}$ descending
3:     $\mathcal{R} \leftarrow \{\}$                                ▷ $\mathcal{R}$ is an ordered set
4:     for $1, \ldots, k$ do
5:         append($\mathcal{R}, \{\}$)                ▷ append a new empty set into $\mathcal{R}$
6:     end for
7:     $r_{min} \leftarrow 0$                   ▷ initialize current minimum sum $r_{min}$
8:     $c_{\mathcal{S}} \leftarrow |\mathcal{S}|$                            ▷ $c_{\mathcal{S}}$ element count of $\mathcal{S}$
9:     for $i \leftarrow 0, \ldots,$ do
10:        for $j \leftarrow 0, \ldots, k-1 \wedge i < c_{\mathcal{S}}$ do
11:            if sum($\mathcal{R}_j$) $\leq s_{min}$ then
12:                insert($\mathcal{R}_j, x$)              ▷ insert $x$ into set at $\mathcal{R}_j$
13:                $r_{min} \leftarrow$ sum($\mathcal{R}_j$)          ▷ set $r_{min}$ to sum of $\mathcal{R}_j$
14:                $i \leftarrow i + 1$
15:            end if
16:        end for
17:    end for
18:    return $\mathcal{R}$
19: end
```

- Additional cluster parameters like density could be taken into account to achieve an improved estimate.
- The actual workload can also be measured during simulation runtime. This measurement may then be used to rate the load for the next simulation run.

2.5.3 Conclusion

Of the general load balancing concepts discussed in this section the best suited approach, for balancing the dynamically distributed game physics simulation, is a centralized global dynamic load balancing algorithm. Centralized because the main game logic is already in a single place. Global because the distributed system is relatively small and it makes no sense to define neighborhoods between the processing nodes when there is no network structure that already provides such relationships. Dynamic because the workload may change, while the simulation is running.

Chapter 3

Implementation

The following chapter describes the testing application developed during the creation of this thesis.

The application facilitates a distributed game library, which aims to run on multiple *Raspberry Pi* (see Section 3.1) microcomputers in parallel. The library was expanded to support distributed physics simulation as proposed in Chapter 1. It incorporates the techniques described in Chapter 2. The computers are connected to a *LAN* through *Ethernet* forming the distributed hardware platform on which the application runs (see Fig. 3.1).

The term node or processing node is used in the following to name a single computer in the distributed system and also for the software running on such computer. Likewise, the term distributed system addresses either the hardware or the software configuration forming the platform that runs the game.

With this engine as a base, a testing application was created. This application is a 2D physics playground where different test scenarios can be started, and a user can interact with simulated objects.

Test results of how the application performs at runtime on different hardware setups are presented and analyzed in Chapter 4.

3.1 Target Platform

The target platform of the proposed game engine is a system on a chip (SoC) single board microcomputer. To be more specific a *Raspberry Pi* [39]. Its SoC a *Broadcom BCM2835* combines all the basic PC components in one Chip which sits on on a credit card sized board. The Hardware in detail is composed of:

- a 700 MHz ARM CPU,
- a *Broadcom* VideoCore IV GPU,
- 256 MB (Model A) or 512 MB RAM (Model B, see Fig. 3.2) and

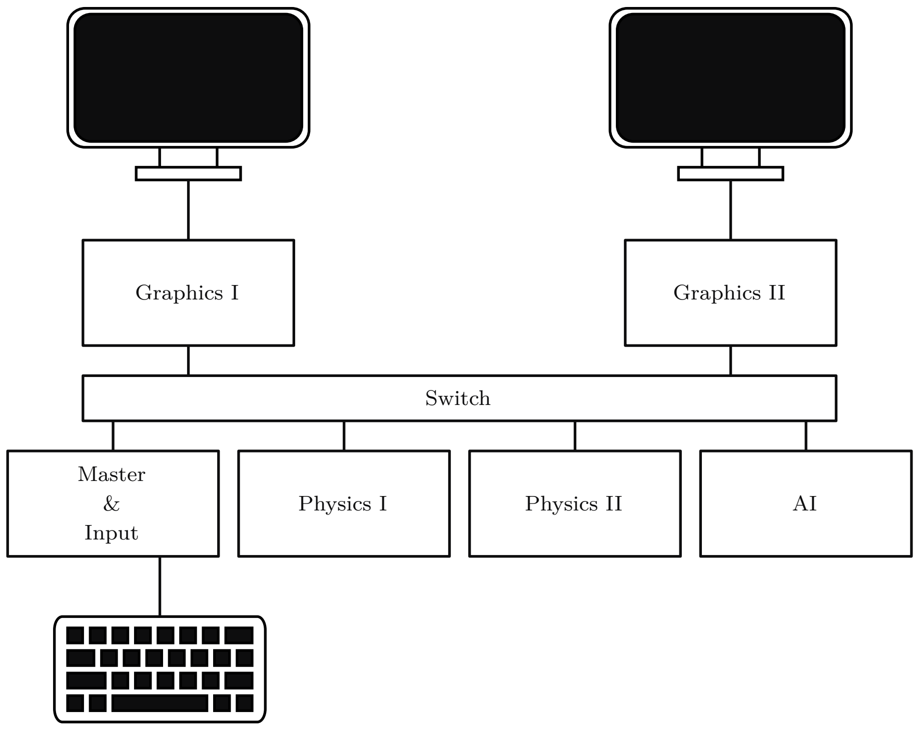


Figure 3.1: An example configuration of the distributed game engine. Six *Raspberry Pis* form a *LAN* by connecting to a network switch. The master node in this case also runs the input module to handle key events of the connected keyboard. To support two high definition screens two *Raspberry Pis* run an instance of the graphics module. An *AI* module and two physics simulation modules run on the remaining *Raspberry Pis*.

- a LAN9512 USB 2.0 Hub and 10/100 Ethernet.

There are also several special hardware interfaces to directly connect devices like cameras, displays or sensors. These are the *GPIO* (General Purpose Input/Output) pins, the *CSI* (Camera Serial Interface) and the DSI (Display Serial Interface). Important for distributed systems of *Ethernet* connected *Raspberry Pi*s is also the maximum network speed of 100Mbps (MegaBits per second).

A custom *Debian Linux* based operating system called *Raspian*[1] is provided for it. Other operating systems like *Fedora Linux*, *Risc OS*, *FreeBSD* and *Plan 9* have also been ported to this hardware platform [38]. Because of its low cost and low power requirements it has already been used in several research projects [1, 27].

[1]https://www.raspbian.org/.

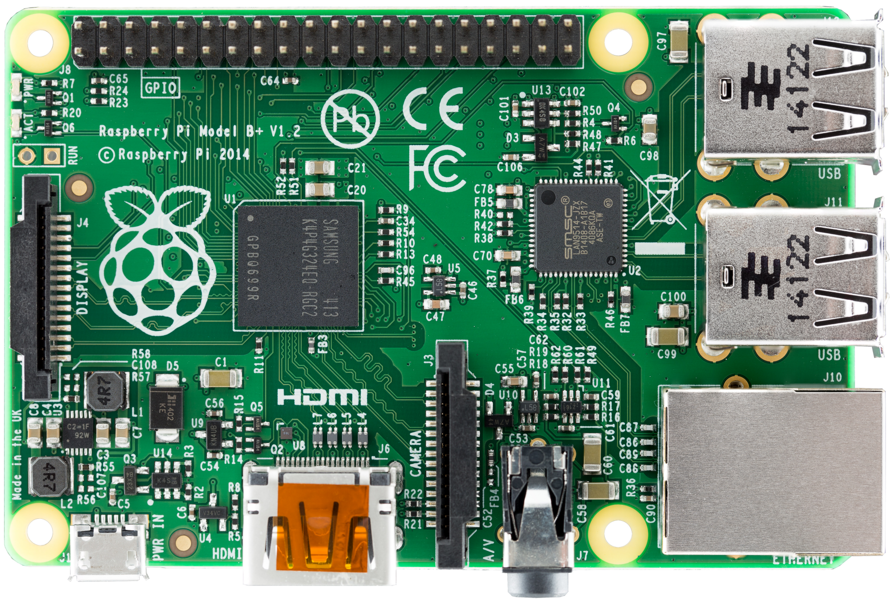


Figure 3.2: Top view of a *Raspberry Pi* Model B+. Four *USB* ports and the ethernet port occupy the right border of the board. *HDMI*, micro-*USB* and audio port are at the bottom border. At the top left the *GPIO* pins are visible. Image source [38].

3.1.1 Operating System

The distributed game engine specifically targets the *Raspbian* OS[2]. Although this OS comes with an *X11*[3] desktop environment (see Fig. 3.3) the game engine is a CLI[4] application. The decision to not require *X11* was made due to its overhead, which may be a problem running games on the low performance hardware of the *Raspberry Pi*.

3.2 Distributed Game Engine

The distributed game engine is the library which allows the creation of games running on one or more *Raspberry Pis* connected to a LAN via Ethernet.

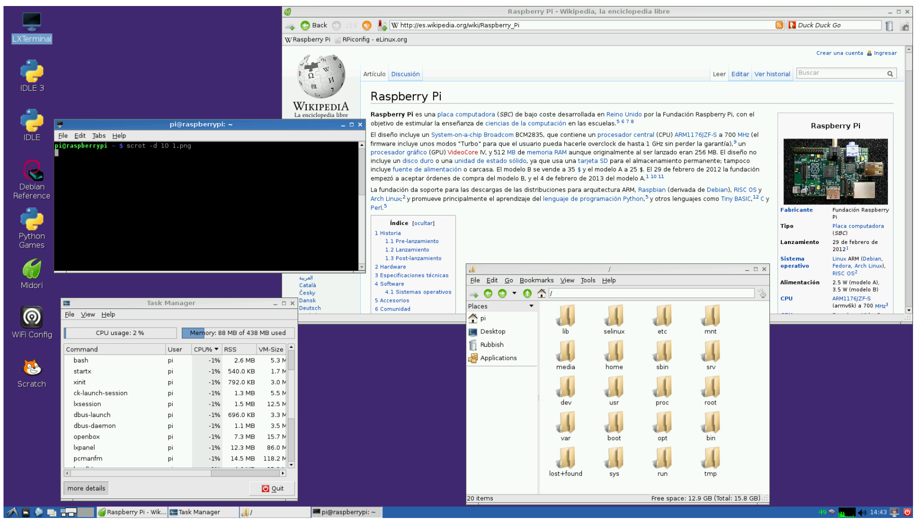
3.2.1 Overview

The engine and the test application were implemented in C++ using the new features introduced by the C++11 standard. Both are designed in a modular

[2] *Operating System.*

[3] *X Window System* http://www.x.org/.

[4] *Command Line Interface.*



```
pi@raspberrypi ~ $ scrot -d 10 1.png
```

Raspberry Pi es una placa computadora (*SBC*) de bajo coste desarrollada en Reino Unido por la Fundación Raspberry Pi, con el objetivo de estimular la enseñanza de ciencias de la computación en las escuelas.[5] [6] [7] [8]

El diseño incluye un System-on-a-chip Broadcom BCM2835, que contiene un procesador central (CPU) ARM1176JZF-S a 700 MHz (el firmware incluye unos modos "Turbo" para que el usuario pueda hacerle overclock de hasta 1 GHz sin perder la garantía),[9] un procesador gráfico (GPU) VideoCore IV, y 512 MB de memoria RAM aunque originalmente al ser lanzado eran 256 MB. El diseño no incluye un disco duro o una unidad de estado sólido, ya que usa una tarjeta SD para el almacenamiento permanente; tampoco incluye fuente de alimentación o carcasa. El modelo B se vende a 35 $ y el modelo A a 25 $. El 29 de febrero de 2012 la fundación empezó a aceptar órdenes de compra del modelo B, y el 4 de febrero de 2013 del modelo A.[1] [10] [11]

La fundación da soporte para las descargas de las distribuciones para arquitectura ARM, Raspbian (derivada de Debian), RISC OS y Arch Linux;[2] y promueve principalmente el aprendizaje del lenguaje de programación Python,[5] y otros lenguajes como Tiny BASIC,[12] C y Perl.[5]

Índice [ocultar]
1 Historia
1.1 Pre-lanzamiento
1.2 Lanzamiento
1.3 Post-lanzamiento
2 Hardware
3 Especificaciones técnicas
4 Software
4.1 Sistemas operativos
5 Accesorios
6 Comunidad

Raspberry Pi

Fabricante	Fundación Raspberry Pi
Tipo	Placa computadora (*SBC*)
Lanzamiento	29 de febrero de 2012[1]
Sistema operativo	Linux ARM (Debian, Fedora, Arch Linux), RISC OS[2]
Alimentación	2.5 W (modelo A), 3.5 W (modelo B)
CPU	ARM1176JZF-S (armv6k) a 700 MHz[3]

CPU usage: 2 %　Memory: 88 MB of 438 MB used

Command	User	CPU%	RSS	VM-Size
bash	pi	-1%	2.6 MB	5.3 M
startx	pi	-1%	540.0 KB	1.7 M
xinit	pi	-1%	792.0 KB	3.0 M
ck-launch-session	pi	-1%	1.3 MB	5.5 M
lxsession	pi	-1%	1.5 MB	12.5 M
dbus-launch	pi	-1%	696.0 KB	3.3 M
dbus-daemon	pi	-1%	1.1 MB	3.5 M
openbox	pi	-1%	7.3 MB	15.7 M
lxpanel	pi	-1%	12.3 MB	86.0 M
pcmanfm	pi	-1%	14.5 MB	118.2 M

Figure 3.3: Screenshot showing *Raspbian*'s *X11* desktop environment with several applications running. Image source [38]

way. The engine aims to be general purpose engine for any type of game. Therefore, certain parts of it could be adapted when implementing a specific game. The modules can be grouped in the core modules and distributable modules (see Fig. 3.4).

Core Modules

The core modules are mandatory and, therefore, run on each single node of the distributed engine. Parts of these modules may be deactivated in certain configurations:

- The *Time System* that manages the different timers needed in a game.
- A *Message System* which allows loose coupling between all the modules of the engine (see Sec. 3.2.3).
- The *Network Module*, which can be seen as the bottom layer of the engine and provides the functionality to run the engine actually distributed among ethernet connected computing nodes.

Distributable Modules

The five distributable modules are the ones that can run spread among the cluster nodes. They form different distributed systems depending on how they are distributed:

- An *Input Module* which handles keyboard input to allow users to interact with the game world.

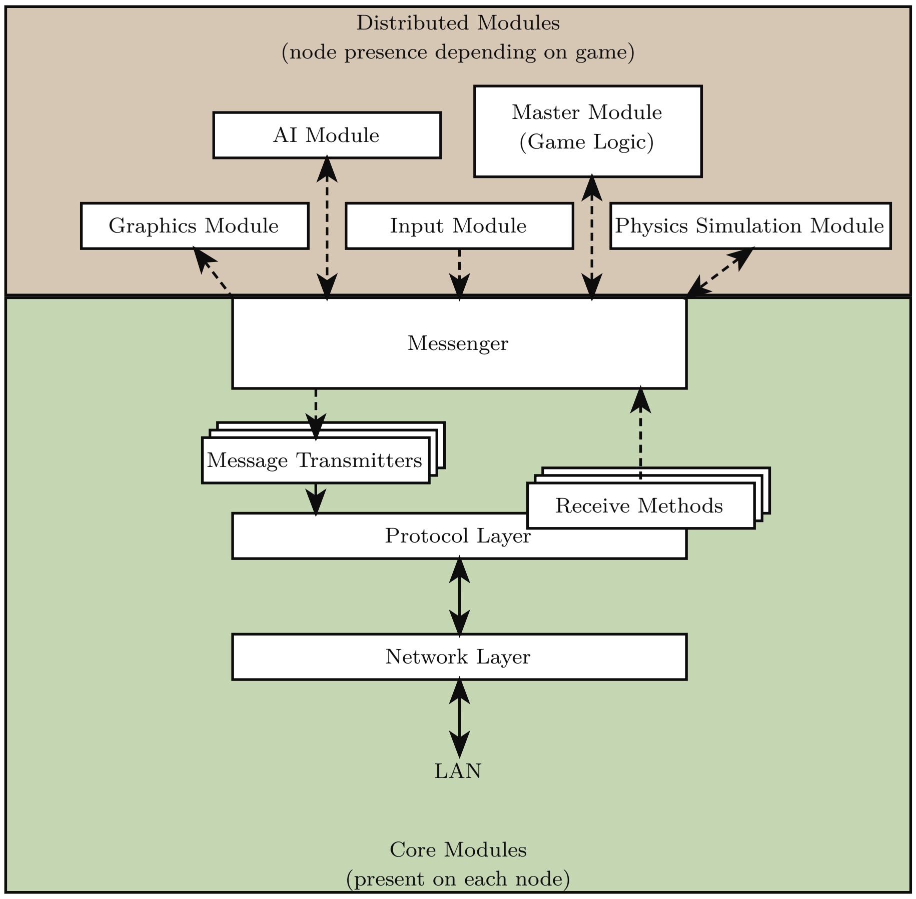


Figure 3.4: Distributed game engine structure. The core modules (green) are present on each single node of the distributed system. Of the distributed modules (brown) presence on a specific node may vary from one to all of them, depending on the game. Also they may be not present in the whole system at all (e.g. no AI module if not needed). There are multiple message transmitters and receive methods for different message categories. The dashed arrows represent the loose coupling via the messenger which may also take place across the network.

- The *Graphics Module*, which is currently the only output module, either runs an *OpenGL*[5] or an *OpenVG*[6] renderer to put a visualization of a game's world to the screen.
- An *AI*[7] *Module*, which can run game specific *AI* code. This module is only a wrapper because an actual *AI* module is game specific.

[5]https://www.khronos.org/opengl/.
[6]https://www.khronos.org/openvg/.
[7]Artificial Intelligence.

- The *Physics Simulation Module*, which simulates physical behaviour of objects inside a game world.
- A *Master Module* is also present in all games. It contains the main game logic, which controls world and object creation, reacts to user input and handles the gameplay. When enabling distributed physics in a game, it also manages clustering of the game objects and balancing the clusters among the physics simulation nodes.

They are optional, and the need for them depends on the requirements of the game itself.

Supported Platforms

As stated prior the engine was initially designed to run on the *Raspberry Pi* and the *Rasbian OS*. Initial tests of distributed game physics showed that the rendering performance on this platform is a major bottleneck when displaying a large number of simulated objects. To overcome this restriction when testing distributed physics performance the engine was extended to support also *OS X*[8], specifically version 10.10 *Yosemite*, as a runtime environment. The use of in general platform independent *C++* code and the modular design of the engine made this extension of support relatively simple. Additional platforms may be supported in the future if needed.

3.2.2 Time System

The time system facilitates functionality contained in the newly introduced standard library `std::chrono` header and namespace in the C++11 standard. To be more specific, the `std::chrono::high_resolution_clock`[9] is used to get time point values in the highest possible precision on a system.

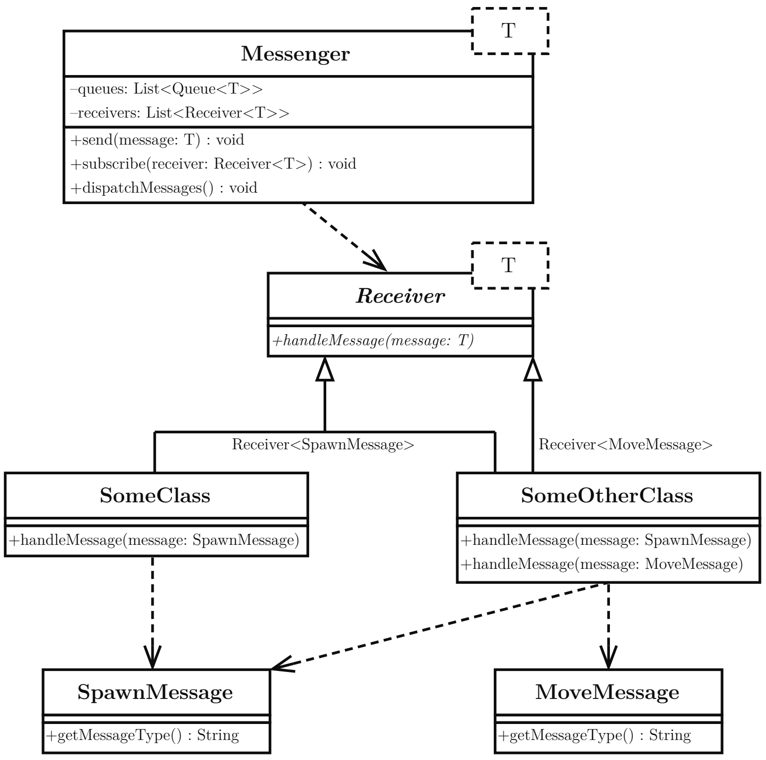
3.2.3 Message System

At the core of the engine lays the message system. It is a generic implementation of the *Event Queue* pattern described in Section 2.1.2. It uses C++ template programming to allow sending and receiving of message objects based on custom message classes. This means if the game needs a particular type of messages a developer simply has to create a class for it. An example *UML* diagram is shown in Figure 3.5.

Message sending then can be achieved by calling the template function of the messenger object. Any class can become a receiver of a specific message type by inheriting from the abstract templateable receiver class for that specific message type. The main benefit is the streamlined and simplified

[8] *OS X (pronounced OS ten).* (OS 10) is the current operating system run on *Apple* computers http://www.apple.com/osx/.

[9] http://en.cppreference.com/w/cpp/chrono/high_resolution_clock



Messenger
T
–queues: List<Queue<T>>
–receivers: List<Receiver<T>>
+send(message: T) : void
+subscribe(receiver: Receiver<T>) : void
+dispatchMessages() : void
Receiver
T
+handleMessage(message: T)
Receiver<SpawnMessage>
Receiver<MoveMessage>
SomeClass
+handleMessage(message: SpawnMessage)
SomeOtherClass
+handleMessage(message: SpawnMessage)
+handleMessage(message: MoveMessage)
SpawnMessage
+getMessageType() : String
MoveMessage
+getMessageType() : String

Figure 3.5: UML diagram showing the templateable messenger and receiver. Basically any class can be used as a message type for this messenger but specialized message classes are more feasible. `SomeClass` and `SomeOtherClass` are only examples of possible extensions of the abstract `Receiver` class. When compared to Figure 2.2 the reduced complexity can be seen in the missing abstract message class.

adding of new messages and receivers and, therefore, less code that could contain errors.

3.2.4 Network Module

The engine's network module provides all the functionality needed to run the engine on multiple nodes. It allows the nodes to discover each other and to later on exchange messages with each other. Since most messages contain information needed at each node (e.g. position updates) they are sent as a broadcast to all nodes in the LAN. To prevent a message loop caused by the broadcast a nodes own message is filtered out during message receiving. If needed certain messages can also be sent by unicast to one node only, which, for example, makes sense for input messages that are only handled at the master node containing the main game logic.

Network Messages

To allow transmission of the message system's messages over the network a serializer and deserializer functionality has to be provided for each message.

A transmitter class for each message category handles serialization of these messages (e.g. `PhysicsTransmitter`). An instance of a transmitter object has to be present on each node that should send out these messages.

Deserialization is implemented as a method within the network modules protocol layer. A call to the message type-specific method is required on each node that is interested in this type.

The instantiation and calls of message serialization and deserialization are handled within the engines modules. For example, the graphics module takes care to receive position updates if it is activated on a node.

3.2.5 Input Module

The input module handles keyboard based player input. It registers key down and key up events. The events are then sent via the messenger to any class extending the according receiver. The module can be configured to handle certain keys only to prevent unnecessary network messages for an unused key.

Input events are read directly from the input device's specific event file (e.g. `/dev/input/event0`) to react to the actual key presses and releases on the *Raspberry Pi* without requiring *X11*. The command-line's default key handling behaviour is disabled during the applications runtime.

On *OS X*, the *GLFW*[10] library is used to receive input events. *GLFW* needs a window to achieve this, so even if the graphics module is not active on an engine node started in *OS X*'s terminal, an empty window is opened. Keyboard events are only registered if this window is active. If the graphics module is active, its window is used for input also.

3.2.6 Graphics Module

The graphics module displays the game world's representation on the screen. To support multiple displays it can run on multiple nodes of one distributed system.

Output can be rendered in a vector mode where polygons are created on a spawn polygon message. This mode is also intended for debugging. In games that use images and sprites for their game world and entity representations, a helper module is needed. This helper module reacts to spawn messages and then triggers the sprite creation in the graphics module. Object movement and destruction are handled in both cases automatically by the graphics module itself.

[10]GLFW is an open source, multi-platform library for creating windows with *OpenGL* contexts and receiving input and events: http://www.glfw.org/.

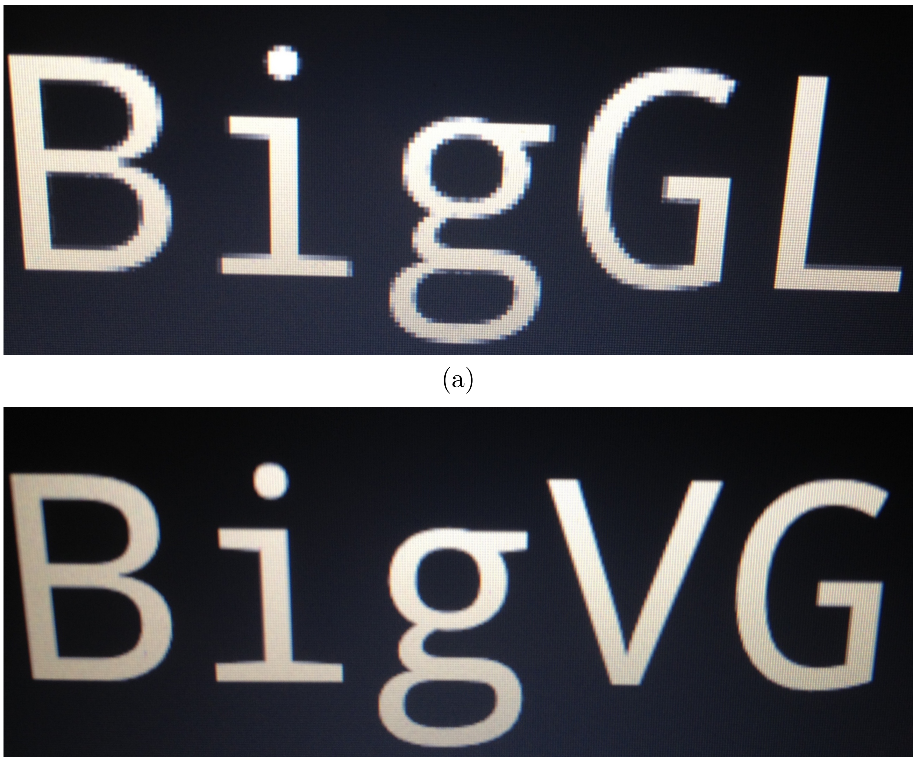


Figure 3.6: Comparison photos taken of the *Raspberry Pi* attached display. Visible is the aliasing at rounded shapes in the *OpenGL* rendering (a) compared to smoother curves in the *OpenVG* rendering (b). Both outputs have the same performance requirements. While the *OpenVG* text (b) is scale-invariant, *OpenGL* text (a) can only be upscaled with quality losses or more demanding higher resolution textures.

On the *Raspberry Pi* the graphics module works without an *X11* environment, which is a speciality when compared to other modern graphics engines. It facilitates low-level screen access via *EGL*[11] and the *Raspberry Pi's* native *Broadcom* graphics driver interface[12]. For the window provided by this base layer either an *OpenGL ES*[13] or an *OpenVG* context is created.

OpenGL ES is a variant of *OpenGL* for embedded systems and the renderer provides functionality to display polygons, images and text. It also supports custom shaders and can be used to create all kinds of graphical effects.

OpenVG, on the other hand, is missing these custom shaders but offers

[11]EGL Native Platform Interface: https://www.khronos.org/egl/.

[12]https://github.com/raspberrypi/firmware/blob/master/opt/vc/include/bcm_host.h.

[13]https://www.khronos.org/opengles/.

more efficient rendering of antialiased polygons especially suited for text (see Fig. 3.6). Displaying raster graphics is also supported. Overall rendering is slightly (up to 10%) more performant than in *OpenGL*.

For *OS X*, the existing render logic was extended to support *OpenGL*. Due to the similarities to *OpenGL ES* the adaption to this platform only needed slight changes in the renderer code and the actual renderers share most of their code. The primary difference is the use of *GLFW* for window and context creation and handling.

3.2.7 AI Module

This module is designed only as a wrapper for a game's AI behaviour. It receives updates about the game world, like spawn and move messages, and sends input events similar to the input module. In games with multiple *AI* opponents, the module can be run on a separate node for each opponent.

3.3 Distributed Physics

All explanations in this chapter are restricted by the fact that a pre-existing physics engine is used. If the physics engine itself was created to be distributed among multiple processing nodes some of the restrictions described may not apply.

Physics simulation in games happens within the game world. To distribute the simulation of the objects in this world the objects need to be clustered by certain requirements:

- Objects that are touching each other need to be simulated together, to allow the physics engine to compute and apply the collision effects.
- Forces that act in specified regions have to be applied to all objects within that region.

The distributed game physics simulation can be structured into several sub-elements. At the core, there is the distributed game engine explained in the previous section. An arbitrary number of computation nodes of this engine may run a physics simulation module. The engines network module handles communication between these separated modules (see Section 3.2.4 for details).

The master node of the engine, which handles the main game logic, is also responsible for distributing the physical objects among the nodes running physics simulation modules. This is done by clustering all objects into groups that need to be simulated on a single node. Then the resulting clusters are distributed among all nodes, to keep the computational stress to the nodes as balanced as possible and achieve optimal performance.

Figure 3.7 shows an overview of this workflow during one single simulation step. In the first run of the game loop (the first one of the simulation

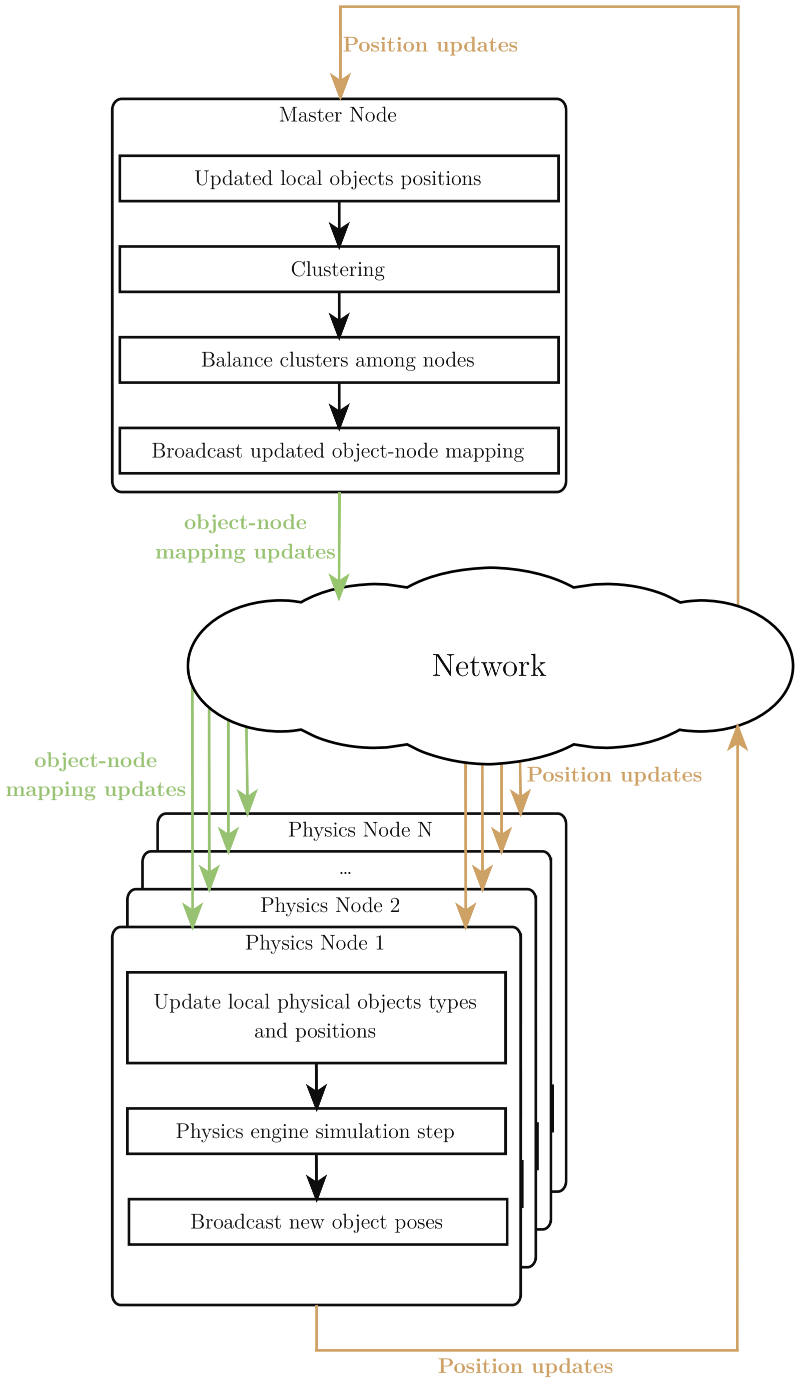

Position updates
Master Node
Updated local objects positions
Clustering
Balance clusters among nodes
Broadcast updated object-node mapping
object-node mapping updates
Network
object-node mapping updates
Position updates
Physics Node N
...
Physics Node 2
Physics Node 1
Update local physical objects types and positions
Physics engine simulation step
Broadcast new object poses
Position updates

Figure 3.7: The workflow of a single step of the distributed game physics simulation during runtime. Communication between the nodes takes place over the network. Not shown in the figure are other node types like graphics nodes and the position updates going there or the input node and the input updates coming from there to the master node.

steps), there are no position updates fed into the master node. Instead the objects initial positions present in the master node are used. This is also true for objects added later during a games runtime. The master module can also run on a node with other modules, for example with a physics simulation module instance.

The following sections offer a detailed look at the parts forming the whole distributed game physics simulation.

3.3.1 Physics Simulation Module

The physics simulation module facilitates a pre-existing third-party physics engine. To keep control over the module's interface wrapper classes for the engines components have been created. This allows to exchange the used physics engine at a later stage if needed and keeps required code changes limited to this wrapper classes.

As stated in Chapter 1, 2D game physics was chosen since it allows simpler setup of the engine testing ground. Reduced complexity of the scenes also makes it easier to compare the test results. For the first implementation of the physics simulation module, the well established Box2D engine was used. Therefore, the wrapper classes are heavily based on the functionality provided by this engine.

Box2D

Box2D is an open source 2-dimensional physics engine for video games developed by Erin Catto and licensed under the *zlib*[14] license [30]. It allows rigid body simulation, is written in platform-independent C++ and was initially released in 2007 [31]. The engine has been ported to other programming languages, including for example *Java*[15], *Adobe Flash*[16], *C#*[17] and *JavaScript*[18,19]. It is incorporated in game engines and frameworks like *Torque2D*[20], *libGDX*[21] or *Unity*[22] and has been used in popular games like *AngryBirds* [32], *Limbo*[23] and *Crayon Physics Deluxe*[24]. Because it's wide adoption, it was chosen to be facilitated for the distributed physics simulation discussed in this thesis.

[14]http://zlib.net/zlib_license.html.
[15]http://www.jbox2d.org/.
[16]http://box2dflash.sourceforge.net/
[17]https://code.google.com/p/box2dx/.
[18]https://code.google.com/p/box2dweb/.
[19]http://box2d-js.sourceforge.net/.
[20]http://www.garagegames.com/community/blogs/view/18641.
[21]https://github.com/libgdx/libgdx/wiki/Box2d.
[22]http://unity3d.com/unity/whats-new/unity-4.3.
[23]https://en.wikipedia.org/wiki/Limbo_(video_game).
[24]https://code.google.com/p/box2d/wiki/FAQ.

The engine simulates rigid body physics and is internally composed of three modules [6]:

1. The *Common* module provides basic functions for memory allocation, math and settings.
2. The *Collision* module defines the shapes, which can be convex polygons, circles or edges and functions that operate on them. It further contains a dynamic tree and broad-phase to allow efficient collision processing of large systems without missing collisions due to the tunneling effect [6, Section 1.5].
3. Finally, the *Dynamics* module handles the simulation world, fixtures that add the previously mentioned shapes to bodies and joints that restrict movement. This module builds on top of the two prior ones, and it is the one that developers interact mostly with when using *Box2D*.

The bodies simulated by the engine can be one of three types:

1. Dynamic bodies are affected by forces like gravity and bounce back when colliding with other bodies. They are used for the player character, other actors and all manipulatable objects in a game scene.
2. Static bodies are not affected by forces and stay in position when dynamic bodies collide with them. They are used for the non-manipulatable game world like floors, walls and so on.
3. Kinematic bodies are like static bodies concerning forces and collisions but can be moved by setting a velocity value for them. A usage example for them is moving platforms in a platform game.

Internally Box2D uses an integrator algorithm to simulate the physics equations at discrete points of time. A time step of at least 60Hz is generally recommended for game physics engines, which is also true for *Box2D*. The lower the update rate the more problems will occur. A higher rate will lead to a more exact simulation. A fixed time step is better than a variable one and, therefore, be used if possible [6, Section 2.4].

3.3.2 Master Module

While the physics simulation module is a wrapper module as it is present in many game engines integrating third-party physics engines. The master module contains the logic to distribute the objects of the game world among the physics simulation module nodes present in the system. This is done as described in the following sections by first clustering physical objects and then balancing this clusters among the nodes depending on their computational load. The process is depicted inside the *Master Node* in Figure 3.7.

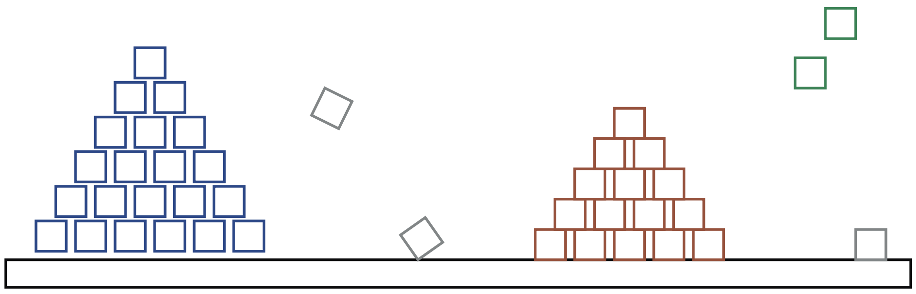


Figure 3.8: Example game scene. The colors (blue, green and red) show the different clusters detected by *DBSCAN*. The grey objects are noise objects not belonging to any cluster. The black bar at the bottom is a static object representing the ground and since only dynamic objects are clustered ignored by the algorithm.

3.3.3 Clustering

For clustering of the physical game objects, the *DBSCAN* algorithm as described in Section 2.4.2 was chosen. The decision to use this algorithm fell for several reasons:

1. It performed better than the other algorithms during comparison tests.
2. Clusters are characterized by the density of objects in an area. This fact fits very well with the requirements to compute objects close to each other together on a single processor node.
3. And at last the free objects not added to any cluster by the algorithm can be distributed arbitrarily in the load balancing step, which allows fine-grained load control in most cases.

Clustering is only done for physical objects composed of dynamic bodies. Static bodies like floors and walls are ignored by the algorithm. On the simulation nodes all objects not simulated locally are also present as static objects.

Algorithm Parameters

As described in Section 2.4.2, *DBSCAN* has two input parameters. They are the minimum number of points in a cluster (`MinPts`) and the radius ϵ around a point within which at least `MinPts` other points are.

The creators of *DBSCAN* developed a heuristics on how to determine these two values for the "thinnest" cluster in a database [8]. However, this approach is only useful for a given database. In the case of the objects inside the game world the database changes in every simulation step. Obtaining the ideal parameters would be too runtime intensive, so a different approach was chosen. The cluster density we are interested in depends on the objects' size, so ϵ is chosen based on the diameter of the largest objects. This way it is guaranteed that all clusters are found in which collisions are possible (see Fig. 3.8).

3.3.4 Load Balancing

Based on the general considerations presented in Section 2.5.2, the final load balancing algorithm was developed in an iterative approach. Workload is managed central because the distributed game engine already uses a master node for the main logic and balanced globally (see Sec. 2.5). The partition of the clusters obtained by *DBSCAN* is done by an implementation of Algorithm 2.5 after classifying the clusters with positive integer values.

Balancing by Cluster Size

The first iteration takes the clusters size as an estimation of the clusters' workload. The size is an integer value larger than two. Noise objects can be seen as clusters containing only one object and, therefore, are weighted by the integer value one.

Balancing by Estimated Cluster Workload

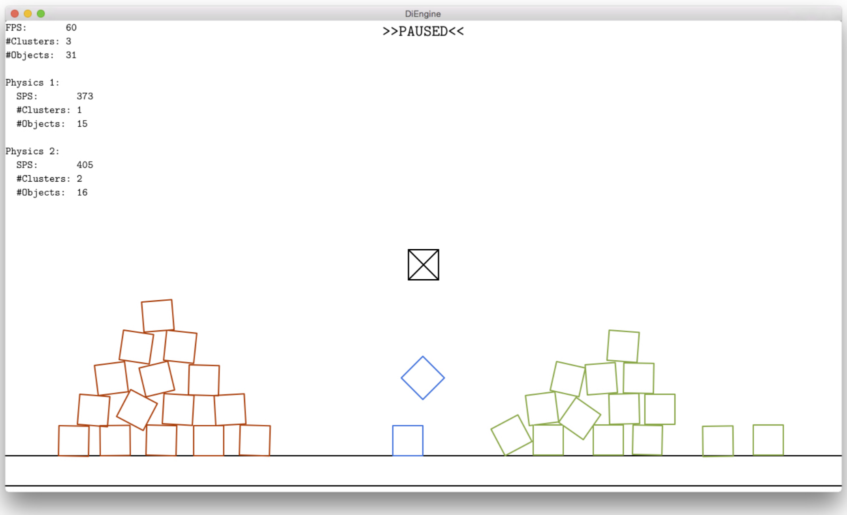
However, clusters containing the same number of objects have not necessarily the same computational load when it comes to physics simulation. A better estimation can be achieved by taking cluster density into account. This approach is based on the assumption that denser clusters more likely have collisions between their objects. The reactions to the collisions have to be calculated during the simulation step and lead to higher workload. The quantification of a clusters load is done by a combination of density and object count.

Balancing by Measured Cluster Workload

In most cases, it is correct to assume that there are more collisions in denser clusters, but there are situations where this is not true. For example Objects in a dense cluster could be moving apart. There is even the possibility that a dense cluster has no load at all which happens if a stack of objects resting on each other is sent to sleep by the physics engine. In the described special cases, the difference between actual and estimated workload is very high, which would lead to an uneven load distribution among the nodes.

Noise objects resting on the ground or not affected by any force may also be sent to sleep by the physics engine. Here a wrongly estimated workload has a much lower effect and will not influence the even balancing as much.

In the first simulation step and if a completely new cluster is formed workload classification falls back to the estimation by density approach since there is no previous workload present that could be measured.



```
FPS:        60
#Clusters: 3
#Objects:  31

Physics 1:
  SPS:        373
  #Clusters: 1
  #Objects:  15

Physics 2:
  SPS:        405
  #Clusters: 2
  #Objects:  16
```

Figure 3.9: Screenshot of the test application showing two stacks of dynamic physical objects and the cursor in interaction mode. The right stack already crumbles to pieces, while the left one remains relatively stable. Below the cursor an object created at the cursor's position is falling. The colors show the different clusters. As expected the two initial stacks form clusters (red and green). In the middle, the player created object and one object originally belonging to the right stack form a third cluster (blue). Basic runtime information is displayed in the top left corner.

Integer values weighted by the measured run time are utilized for partitioning when this method is used.

3.4 Test Application

Using the distributed game engine with the added distributed 2D game physics as the base library a demo and test application was created. The application consists of a simple 2D game world enclosed by a floor and walls on each side.

The player can control an onscreen cursor via keyboard inputs. New game objects can be spawned and destroyed at the cursor's position. The cursor can also mimic a static game object and collide with existing dynamic objects within the virtual environment (see Fig. 3.9). When initially started the game world contains already a bunch of objects. The test application is started by a script on all nodes that compose the distributed system it should run on.

Chapter 4

Results

In this chapter, the results of running the test application described in Section 3.4 in various hardware and software configurations are presented. To get comparable results these configurations have been defined precisely. A photo of an example hardware setup running a test case is shown in Figure 4.1.

4.1 Hardware Setups

To evaluate the implementation the test application was run on three different hardware setups (see Fig. 4.2). The *Master* module contains the whole

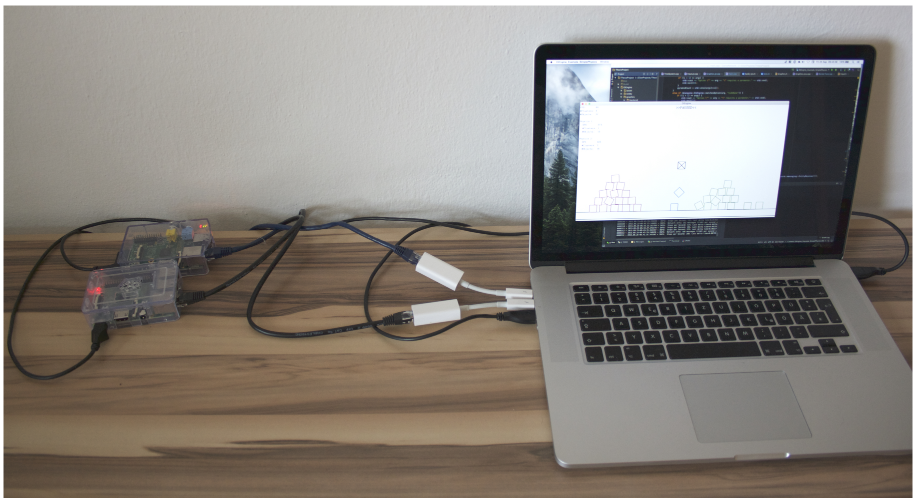


Figure 4.1: Example hardware setup running a test configuration. In this case the notebook running *OS X* also acts as network switch and via USB as power source for the *Raspberry Pi*s.

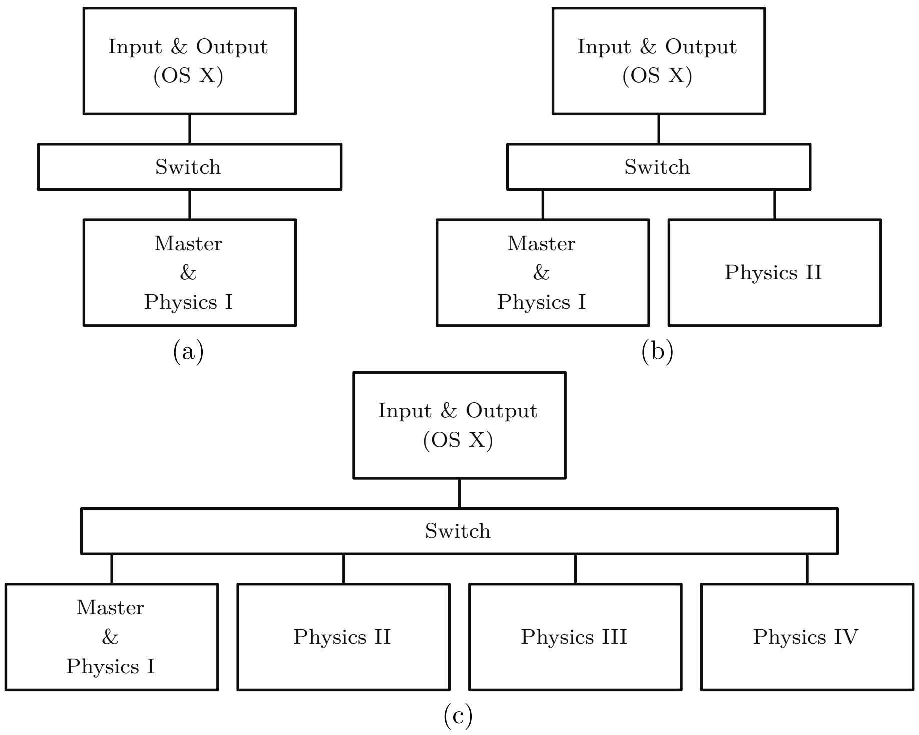


Figure 4.2: The three different hardware setups used for comparison tests. *Setup 1* with non distributed physics simulation running on a single *Raspberry Pi* computer (a), *Setup 2* with physics simulation distributed among two *Raspberry Pi* computers (b), *Setup 3* with physics simulation distributed among four *Raspberry Pi* computers (c). Input and output modules are running on a laptop computer running *OS X* in all setups. Although not needed the switch is also present in test setup 1 to keep the network setup close to the other setups. Setup 2 is also depicted in Figure 4.2.

logic for initial game world setup and handling player inputs sent by the input module. Additionally for the distributed game physics it also takes care of the clustering of the physical objects and after that balancing the clusters depending on their load among the actual physics simulation nodes.

The master node was run on a *Raspberry Pi* node to evaluate the performance effect of clustering and load balancing on the actual target platform. This way the test setup containing only one *Raspberry Pi* can be used to compare the distributed physics simulation performance to a non-distributed physics simulation.

4.1.1 Input and Output

Input and output module are not run on low performance hardware but instead on a notebook running *OS X*. However, varying frame rate on this

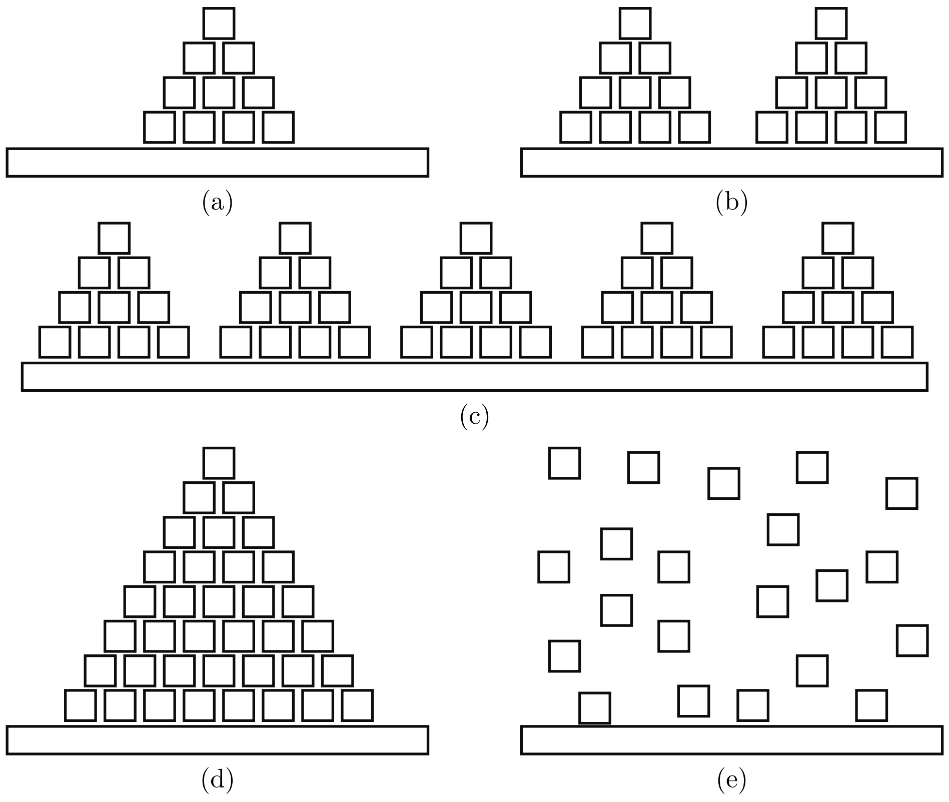


Figure 4.3: Schematic representation of the five game configurations for comparison testing. *Test 1* with 90 objects on a single stack (a), *Test 2* with 90 objects initially distributed among two stacks of 45 objects each (b), *Test 3* with 250 objects initially distributed among five stacks of 50 objects each, *Test 4* is a special case of initially contains a single stack of 300 objects and finally *Test 5* with 300 objects distributed evenly in the game world to form no clusters initially. The actual configurations contain more objects than shown in these figures. All five configurations have their objects starting above the ground and falling down to get initial collision reactions.

node has no effect on the performance of the other nodes that form the distributed physics simulation because it only reacts to the position updates received via the network. Therefore, it could also be replaced with another *Raspberry Pi*, which could render the large number of objects only at very low frame rates, without influencing the behaviour of the other distributed physics simulation.

4.2 Game World Configurations

Five different game world configurations were executed on each one of the three hardware setups. The configurations differ in initial object count and distribution (see Fig. 4.3). To keep computation requirements high throughout the whole runtime the physical properties of the objects (e.g. friction and restitution) were set to values that prevent sleeping of objects in stacks. Sleeping could simply be deactivated in the physics engine, but this way bouncing and sliding objects keep workload high throughout the whole runtime.

To get further performance results, that resemble real world gaming, the game world is manipulated by player interactions in additional test runs.

4.2.1 Data Collection and Comparison Methods

Due to network delay and clock differences on the different computers during runtime leading to non-deterministic behaviour of the *Box2D* physics engine the actual computational load is different in each test run. To minimize this effect on the test results average values of 10 test runs of each configuration were used for comparison.

4.3 Test Results

This section shows the performance measures and results of the test runs. Table 4.1 shows average simulation steps per seconds achieved on each physics node for all game configurations. In general, the results look like expected. The distributed system performs better if there is more than one large cluster. This can be best seen in the results of Test 3.

In the distributed setups Setup 2 and Setup 3, the higher load on the first node which also runs the master module can be seen. The effect, however, is low enough so that this node is not an outlier decreasing the performance of the whole system.

When looking at the numbers for Setup 3 the lower load on the later nodes can be seen in their slightly better performance (e.g. Test 2 with 198, 212, 223, and 218 steps per second). This happens due to the fact that the balancing algorithm distributes the higher loads to the first nodes in most cases.

When testing real-time applications like games, average results are not the most important benchmark. The minimal simulation steps per second are also an interesting figure because this drop in performance is disrupting the user immersion in the game. Table 4.2 shows the minimum and maximum simulation steps per second achieved.

As shown in Figure 4.4 a first performance drop occurs around second five when running the test cases. This decrease happens because the first

Table 4.1: Average simulation steps per second. The column for setup 2 and setup 3 contains the values for all participating nodes.

	Setup 1	*Setup 2*	*Setup 3*
Test 1 (90 objects, 1 stack)	118	129 132	131 136 142 137
Test 2 (90 objects, 2 stacks)	120	169 180	198 212 223 218
Test 3 (250 objects, 5 stacks)	22	39 45	52 62 68 65
Test 4 (300 objects, 1 stack)	18	13 17	12 16 19 21
Test 5 (300 objects)	11	10 13	9 11 14 15

Table 4.2: Minimum and maximum achieved steps per second. The values are an average of all the test runs and for setup 2 and 3 also an average of the nodes participating in the distributed simulation.

	Setup 1 min/max	*Setup 2* min/max	*Setup 3* min/max
Test 1 (90 objects, 1 stack)	81/140	73/184	75/213
Test 2 (90 objects, 2 stacks)	74/162	67/281	64/340
Test 3 (250 objects, 5 stacks)	16/36	21/87	26/115
Test 4 (300 objects, 1 stack)	10/33	9/47	7/53
Test 5 (300 objects)	4/24	6/33	7/47

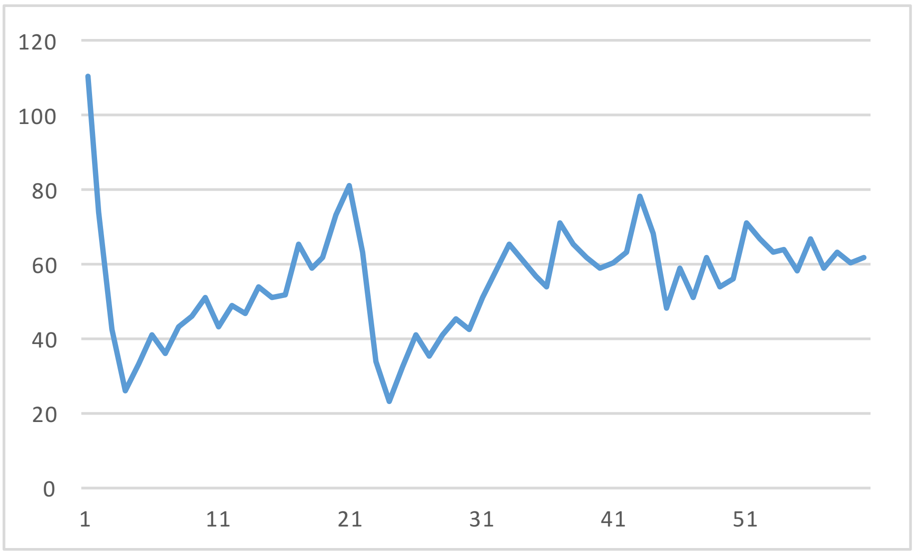


Figure 4.4: Steps per second during the first minute when running Test 3 on Setup 3. The peak at second one exists because there are no collisions initially.

objects hit the ground at this time. Further performance drops can be linked to cluster fluctuations. Without player interaction performance stays mostly at the same level after the first minute of runtime.

4.3.1 Network Performance

Figure 4.5 compares physics simulation performance with network load. Higher network load occurs when more objects are colliding, and larger cluster changes happen. Further factors increasing the network load are the creation of new objects by the player. A higher load can also be seen at simulation start, where object creation messages are sent to all simulation nodes. However, this is compensated by the low amount of collisions at this point.

As seen in Figure 4.6 average network load stays at a relatively low level even for 300 objects. This load is far from exceeding the limit of the 100 Mbps network speed of the *Raspberry Pi*s.

The loads shown in Figure 4.5 and Figure 4.6 are average values. Peaks can occur where the rate reaches the network's limits of 100Mbps. These peaks may lead to a short increase in latency. In the tested setups this latency was never noticeable in the application's output. Maximum measured latency in any of the tests was 78ms which is low enough for most game types (see Sec. 2.2.1).

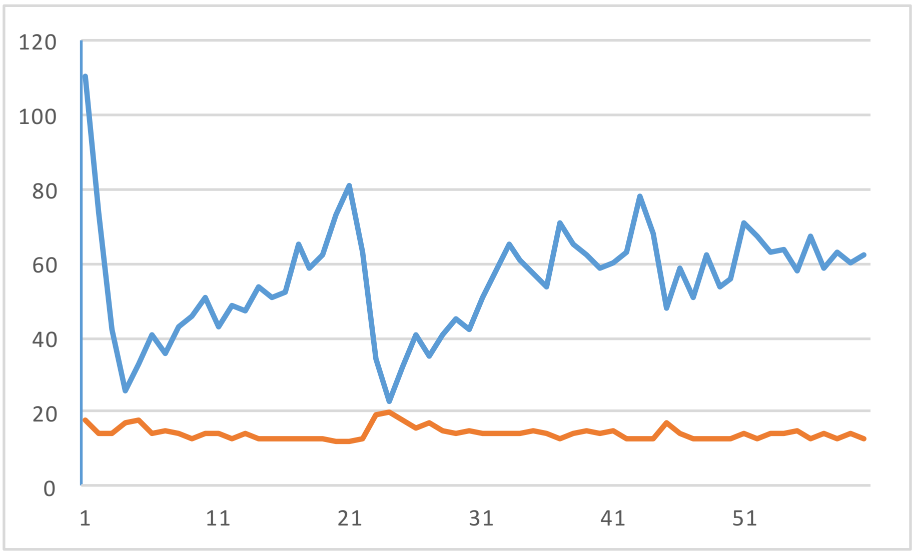


Figure 4.5: The same interval as shown in Figure 4.4 (blue) and the bits per second sent from the *Master Node* (orange). On average 14 bits per second are sent in this scenario.

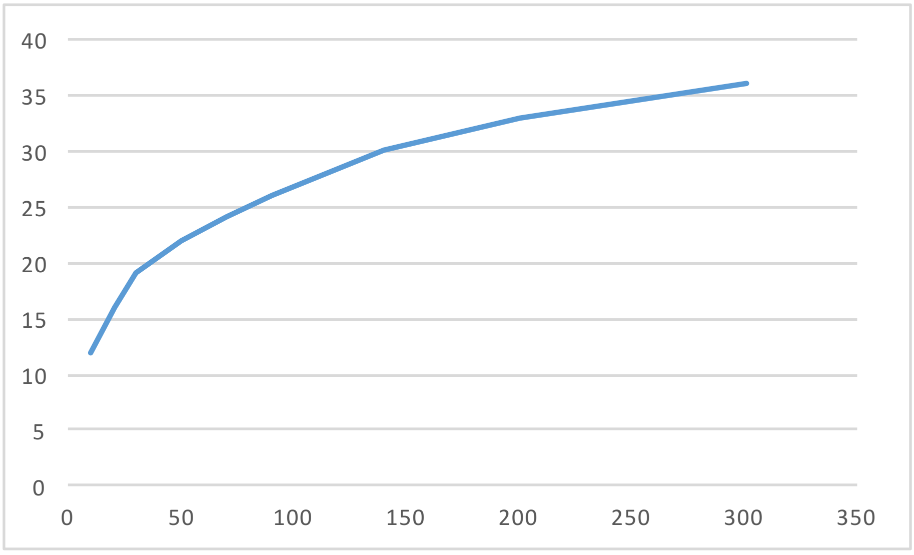


Figure 4.6: Average network load in bits per second (y-axis) depending on numbers of objects to simulate (x-axis) on Setup 3.

4.3.2 Performance Limits

As stated in Section 3.3.1, the minimal recommended time step for *Box2D* is 60Hz. To guarantee this update rate the maximum number of objects is around 70 objects per node. This number also depends highly on the game type and the game world because of different load depending on the interactions occurring between the objects.

Scalability by adding additional nodes is visible in the results shown in Table 4.1. When the peaks in network load are taken into account a system with ten simulation nodes should run without noticeable latency. Therefore, a system of ten simulation nodes should be able to run a simulation of around 700 objects. For larger systems, however, latency and overall network behaviour is not predictable and actual tests are required to give a statement on performance.

These performance limits of the whole distributed system may be lower if the objects form very large clusters that cannot be distributed evenly among the nodes.

Chapter 5

Conclusion

In the present thesis, the creation of a distributed game physics system was shown by introducing the required knowledge base, describing the considerations for the system's creation and evaluating the system in several test scenarios. The goal to improve game performance by distributing physics simulation among multiple devices was achieved to a certain degree as shown in the test results discussed in Chapter 4.

5.1 Problems

The main problems of the generalized implementation presented in this thesis are the special cases. Due to the unpredictability of the players actions all physical objects in a game could be forming a single cluster. Therefore, a distribution among the computation nodes is not permitted anymore. Although this may be a rare case in most games, it is still possible and would lead to significant performance drop.

A situation where a cluster has a very high complexity in regards to the computational requirements can also lead to performance problems. In physics simulations, this can happen in tight object clusters like stacks and heaps. Again the resulting uneven distribution among the processing nodes is the cause of problems.

The current clustering implementation assumes relatively homogenously sized objects. A problem occurs when the sizes differ to a certain degree. Larger objects may not be added to a cluster of smaller objects by the algorithm. Colliding objects that are not computed on the same island have slightly incorrect collision reactions.

5.2 Improvements

As explained in Section 3.2 the existing system is unfocused concerning game types and, therefore, offers a generalized approach to some problems.

However, it could be used as a starting point for more specialized distributed engines and games, which could offer improvements that are not possible in a general approach.

A possible general improvement for the clustering algorithm would be to include the existing velocity information of the objects. This way objects that are close to each other but moving away from each other could be split into multiple clusters. It would further allow to add objects earlier to clusters if they approach them.

Additionally the system could also be adapted to more powerful hardware and adapted for use in online games where the simulation is run on server clusters. The increased performance would allow more complex game worlds containing much more simulated objects but, of course, would need adjustments in the network layer and also lead to different performance bottlenecks.

Appendix A

Content of the CD-ROM

Format: CD-ROM, Single Layer, ISO9660-Format

A.1 Master's Thesis

Pfad: /

Soellinger_Michael_2015.pdf	Master's Thesis (This document)

A.2 Online Sources

Pfad: /Online Sources

*.pdf	Copies of the online sources

A.3 Thesis Project

Pfad: /Thesis Project

DiEngine	Distributed game engine source code
examples/simple_physics	Test application source code
build/pi/bin	*Rasbian* binaries
build/osx/bin	*OS X* 10.10 binaries
CMakeLists.txt	Main *CMake* project file

A.4 Miscellaneous

Pfad: /Images

*.pdf	Vector graphics embedded in the thesis
*.jpg, *.png	Raster graphics embedded in the thesis

References

Literature

[1] Pekka Abrahamsson et al. “Affordable and Energy-Efficient Cloud Computing Clusters. The Bolzano Raspberry Pi Cloud Cluster Experiment”. In: *Proceedings of the 5th International Conference on Cloud Computing Technology and Science*. Bristol, UK: IEEE, Dec. 2013, pp. 170–175 (cit. on p. 18).

[2] Mihael Ankerst et al. “OPTICS. Ordering Points To Identify the Clustering Structure”. In: *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*. Philadelphia, PA, USA: ACM Press, June 1999, pp. 49–60 (cit. on p. 11).

[3] Grenville Armitage, Mark Claypool, and Philip Branch. *Networking and Online Games. Understanding and Engineering Multiplayer Internet Games*. Chichester, UK: Wiley and Sons, Apr. 2006 (cit. on pp. 5, 6).

[4] Quentin Bargard, Anthony Ventresque, and Liam Murphy. “Global Dynamic Load-Balancing for Decentralised Distributed Simulation”. In: *Proceedings of the 2014 Winter Simulation Conference* (Dec. 2014), pp. 3797–3808 (cit. on pp. 13, 14).

[5] Tom Beigbeder et al. “The Effects of Loss and Latency on User Performance in Unreal Tournament 2003”. In: *Proceedings of the 3rd ACM SIGCOMM Workshop on Network and System Support for Games*. NetGames ’04. Portland, OR, USA: ACM, Sept. 2004, pp. 144–151 (cit. on p. 6).

[6] Erin Catto. *Box2D User Manual*. Version 2.3.0. 2013. URL: http://box2d.org/manual.pdf (cit. on p. 29).

[7] Matthias Dick, Oliver Wellnitz, and Lars Wolf. “Analysis of Factors Affecting Players’ Performance and Perception in Multiplayer Games”. In: *Proceedings of the 4th ACM SIGCOMM Workshop on Network and System Support for Games*. NetGames ’05. Hawthorne, NY, USA: ACM, Oct. 2005, pp. 1–7 (cit. on p. 6).

[8] Martin Ester et al. “A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise”. In: *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining*. Portland, OR, USA: AAAI Press, Aug. 1996, pp. 226–231 (cit. on pp. 10–12, 30).

[9] Erich Gamma et al. *Design Patterns. Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addison-Wesley, Oct. 1994 (cit. on p. 4).

[10] Bruce Hendrickson and Karen Devine. “Dynamic Load Balancing in Computational Mechanics”. In: *Computer Methods in Applied Mechanics and Engineering* 184.2–4 (Apr. 14, 2000), pp. 485–500 (cit. on p. 15).

[11] Richard M. Karp. “Reducibility among Combinatorial Problems”. In: *Complexity of Computer Computations*. Ed. by Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger. New York, NY, USA: Springer US, Mar. 1972, pp. 85–103 (cit. on p. 14).

[12] Leonard Kaufman and Peter J. Rousseeuw. *Finding Groups in Data. An Introduction to Cluster Analysis*. Chichester, UK: Wiley and Sons, Jan. 1990 (cit. on p. 10).

[13] Michael Lewis and Jeffrey Jacobson. “Game Engines in Scientific Research”. In: *Communications of the ACM* 45 (January 2002), pp. 27–31 (cit. on pp. 3, 4).

[14] Ian Millington. *Game Physics Engine Development. How to Build a Robust Commercial-Grade Physics Engine for Your Game*. 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Sept. 2010 (cit. on pp. 8, 9).

[15] Bob Nystrom. *Game Programming Patterns*. Genever Benning, Nov. 2014 (cit. on pp. 4, 5).

[16] Lothar Pantel and Lars C. Wolf. “On the Impact of Delay on Real-time Multiplayer Games”. In: *Proceedings of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video*. NOSSDAV ’02. Miami, FL, USA: ACM, May 2002, pp. 23–29 (cit. on p. 7).

[17] Jon Postel. *Internet Protocol. RFC 791. Updated by RFCs 1349, 2474*. Sept. 1981. URL: http://tools.ietf.org/html/rfc791 (cit. on p. 7).

[18] Jon Postel. *Transmission Control Protocol. RFC 793. Updated by RFCs 1122, 3168, 6093, 6528*. Sept. 1981. URL: http://tools.ietf.org/html/rfc793 (cit. on p. 7).

[19] Jon Postel. *User Datagram Protocol. RFC 768*. Aug. 1980. URL: http://tools.ietf.org/html/rfc768 (cit. on p. 7).

[20] Peter Quax et al. “Objective and Subjective Evaluation of the Influence of Small Amounts of Delay and Jitter on a Recent First Person Shooter Game”. In: *Proceedings of 3rd ACM SIGCOMM Workshop on Network and System Support for Games*. NetGames ’04. Portland, OR, USA: ACM, Sept. 2004, pp. 152–156 (cit. on p. 6).

[21] Nathan Sheldon et al. “The Effect of Latency on User Performance in Warcraft III”. In: *Proceedings of the 2nd Workshop on Network and System Support for Games*. NetGames ’03. Redwood City, CA, USA: ACM, May 2003, pp. 3–14 (cit. on p. 6).

[22] Sandeep K. Singhal. *Effective Remote Modeling in Large-Scale Distributed Simulation and Visualization Environments*. Tech. rep. CS-TR-96-1574. Stanford, CA, USA: Stanford University. Department of Computer Science, Aug. 1996. URL: http://i.stanford.edu/pub/cstr/reports/cs/tr/96/1574/CS-TR-96-1574.ps (cit. on p. 7).

[23] Steven S. Skiena. *The Algorithm Design Manual*. 2nd ed. London, UK: Springer-Verlag London, 2008 (cit. on p. 14).

[24] Jouni Smed and Harri Hakonen. *Algorithms and Networking. for Computer Games*. Chichester, UK: John Wiley and Sons, Apr. 2006 (cit. on p. 6).

[25] Jouni Smed, Timo Kaukoranta, and Harri Hakonen. “Aspects of Networking in Multiplayer Computer Games”. In: *The Electronic Library* 20.2 (2002), pp. 87–97 (cit. on p. 6).

[26] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Boston, MA, USA: Addison-Wesley, May 2, 2005 (cit. on p. 11).

[27] Fung Po Tso et al. “The Glasgow Raspberry Pi Cloud. A Scale Model for Cloud Computing Infrastructures”. In: *Proceedings of the 33rd International Conference on Distributed Computing Systems Workshops*. Philadelphia, PA, USA: IEEE, July 2013, pp. 108–112 (cit. on p. 18).

[28] Thomas Y. Yeh, Petros Faloutsos, and Glenn Reinman. “Enabling Real-time Physics Simulation in Future Interactive Entertainment”. In: *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*. Sandbox ’06. Boston, MA, USA: ACM, July 2006, pp. 71–81 (cit. on p. 9).

[29] Thomas Y. Yeh et al. “ParallAX: An Architecture for Real-time Physics”. In: *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ISCA ’07. San Diego, CA, USA: ACM, June 2007, pp. 232–243 (cit. on p. 9).

Online sources

[30] *About | Box2D*. URL: http://www.box2d.org/about/ (visited on 09/24/2015) (cit. on p. 28).

[31] *Box2D - Wikipedia, the free encyclopedia*. URL: https://en.wikipedia.org/wiki/Box2D (visited on 09/24/2015) (cit. on p. 28).

[32] *Creator Of Angry Birds' Physics Engine Calls Out Rovio For Not Giving Him Credit | TechCrunch*. URL: http://techcrunch.com/2011/02/28/creator-of-angry-birds-physics-engine-calls-out-rovio-for-not-giving-him-credit/ (visited on 09/24/2015) (cit. on p. 28).

[33] *Event Loop - Wikipedia, the free encyclopedia*. URL: http://en.wikipedia.org/wiki/Event_loop (visited on 09/24/2015) (cit. on p. 4).

[34] Glenn Fiedler. *Gaffer on Games | Reliability and Flow Control*. Oct. 2008. URL: http://gafferongames.com/networking-for-game-programmers/reliability-and-flow-control/ (visited on 09/24/2015) (cit. on p. 7).

[35] Glenn Fiedler. *Gaffer on Games | UDP vs TCP*. Oct. 2008. URL: http://gafferongames.com/networking-for-game-programmers/udp-vs-tcp/ (visited on 09/24/2015) (cit. on p. 7).

[36] Glenn Fiedler. *Gaffer on Games | Virtual Connection over UDP*. Oct. 2008. URL: http://gafferongames.com/networking-for-game-programmers/virtual-connection-over-udp/ (visited on 09/24/2015) (cit. on p. 7).

[37] *Partition Problem - Wikipedia, the free encyclopedia*. URL: https://en.wikipedia.org/wiki/Partition_problem (visited on 09/24/2015) (cit. on pp. 14, 15).

[38] *Raspberry Pi - Wikipedia*. URL: http://en.wikipedia.org/wiki/Raspberry_Pi (visited on 09/24/2015) (cit. on pp. 18–20).

[39] *What is a Raspberry Pi?* URL: http://www.raspberrypi.org/help/what-is-a-raspberry-pi/ (visited on 09/24/2015) (cit. on p. 17).